目錄

Spring Framework 5 中文文档

本项目为Spring 5 Reference的翻译作品，由并发编程网的同学一起合作而成。

Git Book 在线阅读地址

待整理列表：

- 18. Web MVC 框架

待办：

1. 完成“待整理列表”
2. 文档中代码高亮
3. 文档中加入评论功能，方便反馈文档错误

Contributors

Spring 16:EthanPark

Spring 26:JasonGeng

> 希望读者多多支持,有错误之处及时增加pull request进行修正

1.Spring入门指南

本参考指南提供了有关Spring Framework的详细信息。它全面的介绍了Spring的所有的功能，以及Spring涉及的基础概念（如“依赖注入”“Dependency Injection”）。

如果你是刚开始使用Spring，你可能需要首先创建一个Spring Boot应用程序来开始Spring框架之旅。Spring Boot提供了一个快速（和自治的）的方式来创建一个基于Spring的生产环境。它是基于Spring框架，支持约定优于配置，并且被设计成尽可能快地让你启动和运行程序。

您可以使用start.spring.io生成一个基本项目或按照新手入门指南里的任意一个指南构建项目，例如构建一个RESTful Web服务入门指南 。为了更容易帮助你理解，这些指南都是面向任务的，其中大部分都是基于Spring Boot的。 他们还涵盖了很多Spring原型工程，在您需要解决特定问题时可以考虑使用他们。

2.介绍Spring框架

Spring 框架是一个Java平台，它为开发Java应用程序提供全面的基础架构支持。Spring负责基础架构，因此您可以专注于应用程序的开发。

Spring可以让您从“plain old Java objects”（POJO）中构建应用程序和通过非侵入性的POJO实现企业应用服务。此功能适用于Java SE的编程模型，全部的或部分的适应Java EE模型。

这些例子告诉你，作为一个应用程序开发人员，如何从Spring平台中受益：

- 写一个Java方法执行数据库事务，而无需处理具体事务的APIs。
- 写一个本地Java方法去远程调用，而不必处理远程调用的APIs。
- 写一个本地Java方法实现管理操作，而不必处理JMX APIs。
- 写一个本地Java方法实现消息处理，而不必处理JMS APIs。

2.1依赖注入和控制反转

Java应用程序-这是一个宽松的术语，它包括的范围从受限的嵌入式应用程序到n层的服务器端企业应用程序-通常组成程序的对象互相协作而构成正确的应用程序。因此，在一个应用程序中的对象彼此具有依赖关系（*dependencies*）。

虽然Java平台提供了丰富的应用程序开发功能，但它缺乏将基本的模块组织成一个整体的方法，而将该任务留给了架构师和开发人员。虽然你可以使用如工厂，抽象工厂，*Builder*，装饰器和*Service Locator*等 设计模式来构建各种类和对象实例，使他们组合成应用程序，但这些模式无非只是：最佳实践赋予的一个名字，以及这是什么样的模式，应用于哪里，它能解决的问题等等。 模式是您必须在应用程序中自己实现的形式化的最佳实践。

Spring框架控制反转（IOC）组件通过提供一系列的标准化的方法把完全不同的组件组合成一个能够使用的应用程序来解决这个问题。Spring框架把形式化的设计模式编写为优秀的对象，你可以容易的集成到自己的应用程序中。许多组织和机构使用Spring框架，以这种方式(使用Spring的模式对象)来设计健壮的，可维护的应用程序。

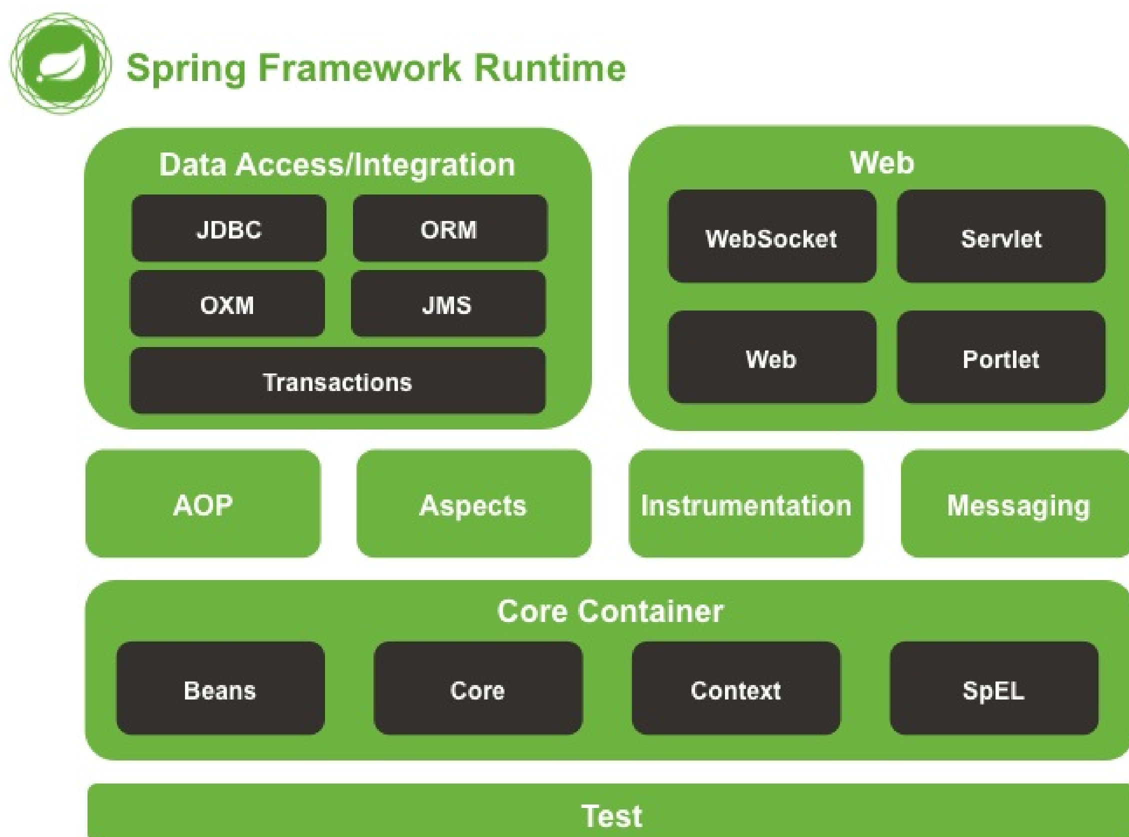
背景

“ 现在的问题是，什么方面的控制被（他们）反转了？ ”马丁·福勒2004年在他的网站提出了这个有关控制反转（IOC）的问题 ，福勒建议重命名，使之能够自我描述，并提出了依赖注入(*Dependency Injection*)。

2.2模块

Spring框架的功能被有组织的分散到约20个模块中。这些模块分布在核心容器，数据访问/集成，Web，AOP（面向切面的编程），植入(Instrumentation)，消息传输和测试，如下面的图所示。

图*2.1 Spring*框架概述



以下部分列出了每个可用模块，以及它们的工件名称和它们支持的主要功能。工件的名字对应的是

工件标识符，使用在依赖管理工具中。

2.2.1核心容器

核心容器由以下模块组成，spring-core， spring-beans，spring-context，spring-context-support，和spring-expression （Spring表达式语言）。

spring-core和spring-beans模块提供了框架的基础功能，包括IOC和依赖注入功能。 BeanFactory是一个成熟的工厂模式的实现。你不再需要编程去实现单例模式，允许你把依赖关系的配置和描述从程序逻辑中解耦。

上下文（spring-context）模块建立在由*Core和Beans*模块提供的坚实的基础上：它提供一个框架式的对象访问方式，类似于一个JNDI注册表。上下文模块从Beans模块继承其功能，并添加支持国际化（使用，例如，资源集合），事件传播，资源负载，并且透明创建上下文，例如，Servlet容器。Context模块还支持Java EE的功能，如EJB，JMX和基本的远程处理。ApplicationContext接口是Context模块的焦点。 spring-context-support支持整合普通第三方库到Spring应用程序上下文，特别是用于高速缓存（ehcache，JCache）和调度（CommonJ，Quartz）的支持。

spring-expression模块提供了强大的表达式语言去支持查询和操作运行时对象图。这是对JSP 2.1规范中规定的统一表达式语言（unified EL）的扩展。该语言支持设置和获取属性值，属性分配，方法调用，访问数组，集合和索引器的内容，逻辑和算术运算，变量命名以及从Spring的IoC容器中以名称检索对象。 它还支持列表投影和选择以及常见的列表聚合。

2.2.2 AOP和Instrumentation

spring-aop模块提供了一个符合*AOP*联盟（要求）的面向方面的编程实现，例如，允许您定义方法拦截器和切入点（pointcuts），以便干净地解耦应该被分离的功能实现。 使用源级元数据(source-level metadata)功能，您还可以以类似于.NET属性的方式将行为信息合并到代码中。

单独的spring-aspects模块，提供了与AspectJ的集成。

spring-instrument模块提供了类植入(instrumentation)支持和类加载器的实现,可以应用在特定的应用服务器中。该spring-instrument-tomcat 模块包含了支持Tomcat的植入代理。

2.2.3消息

Spring框架4包括spring-messaging(消息传递模块)，其中包含来自Spring Integration的项目，例如，Message，MessageChannel，MessageHandler，和其他用来传输消息的基础应用。该模块还包括一组用于将消息映射到方法的注释(annotations)，类似于基于Spring MVC注释的编程模型。

2.2.4数据访问/集成

数据访问/集成层由JDBC，ORM，OXM，JMS和事务模块组成。

spring-jdbc模块提供了一个JDBC –抽象层，消除了需要的繁琐的JDBC编码和数据库厂商特有的错误代码解析。

spring-tx模块支持用于实现特殊接口和所有POJO（普通Java对象）的类的编程和声明式事务 管理。

spring-orm模块为流行的对象关系映射(object-relational mapping)API提供集成层，包括JPA和Hibernate。使用spring-orm模块，您可以将这些O / R映射框架与Spring提供的所有其他功能结合使用，例如前面提到的简单声明性事务管理功能。

spring-oxm模块提供了一个支持对象/ XML映射实现的抽象层，如JAXB，Castor，JiBX和XStream。

spring-jms模块(Java Messaging Service) 包含用于生产和消费消息的功能。自Spring Framework 4.1以来，它提供了与 spring-messaging模块的集成。

2.2.5 Web

Web层由spring-web，spring-webmvc和spring-websocket 模块组成。

spring-web模块提供基本的面向Web的集成功能，例如多部分文件上传功能，以及初始化一个使用了Servlet侦听器和面向Web的应用程序上下文的IoC容器。它还包含一个HTTP客户端和Spring的远程支持的Web相关部分。

spring-webmvc模块（也称为Web-Servlet模块）包含用于Web应用程序的Spring的模型-视图-控制器(*MVC*)和REST Web Services实现。 Spring的MVC框架提供了领域模型代码和Web表单之间的清晰分离，并与Spring Framework的所有其他功能集成。

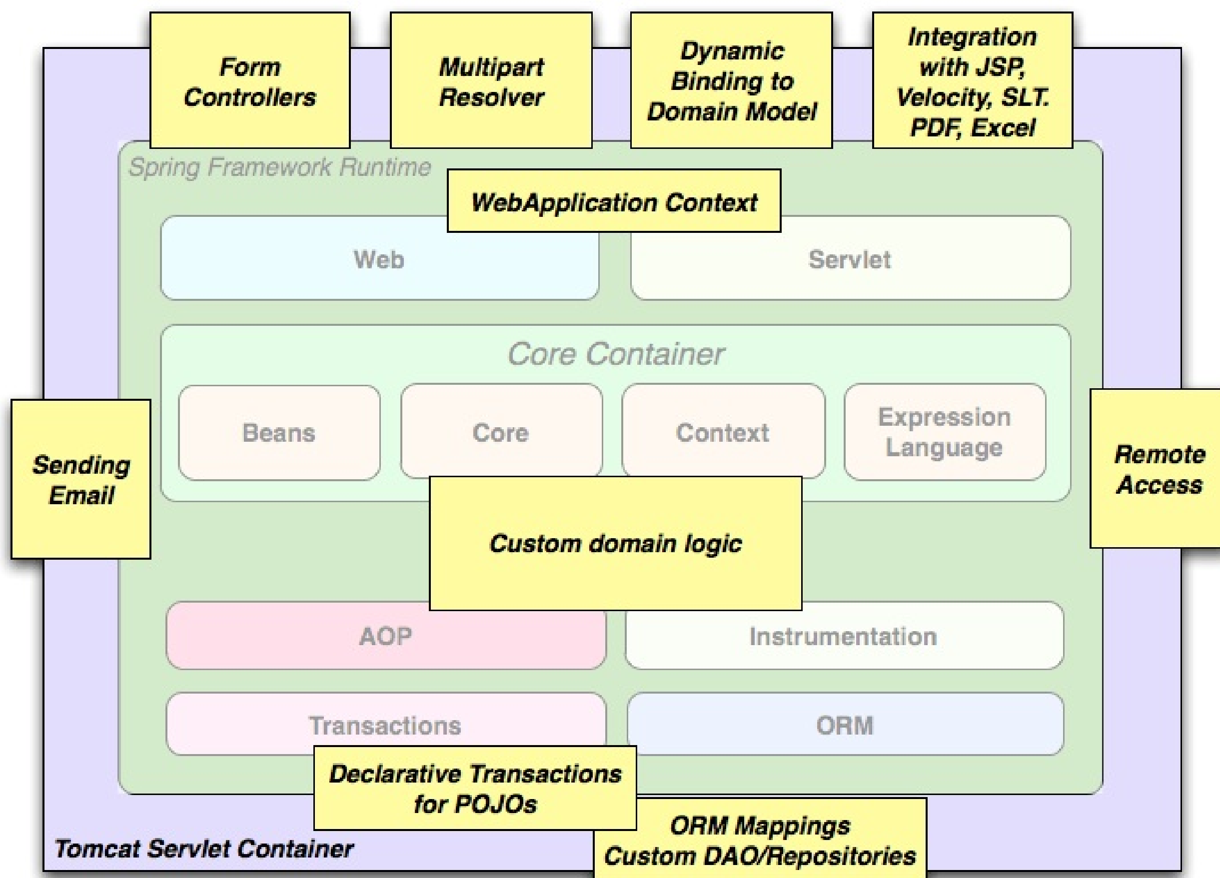
2.2.6测试

spring-test模块支持使用JUnit或TestNG对Spring组件进行单元测试和 集成测试。它提供了Spring ApplicationContexts的一致加载和这些上下文的缓存。它还提供可用于独立测试代码的模仿(mock)对象。

2.3使用场景

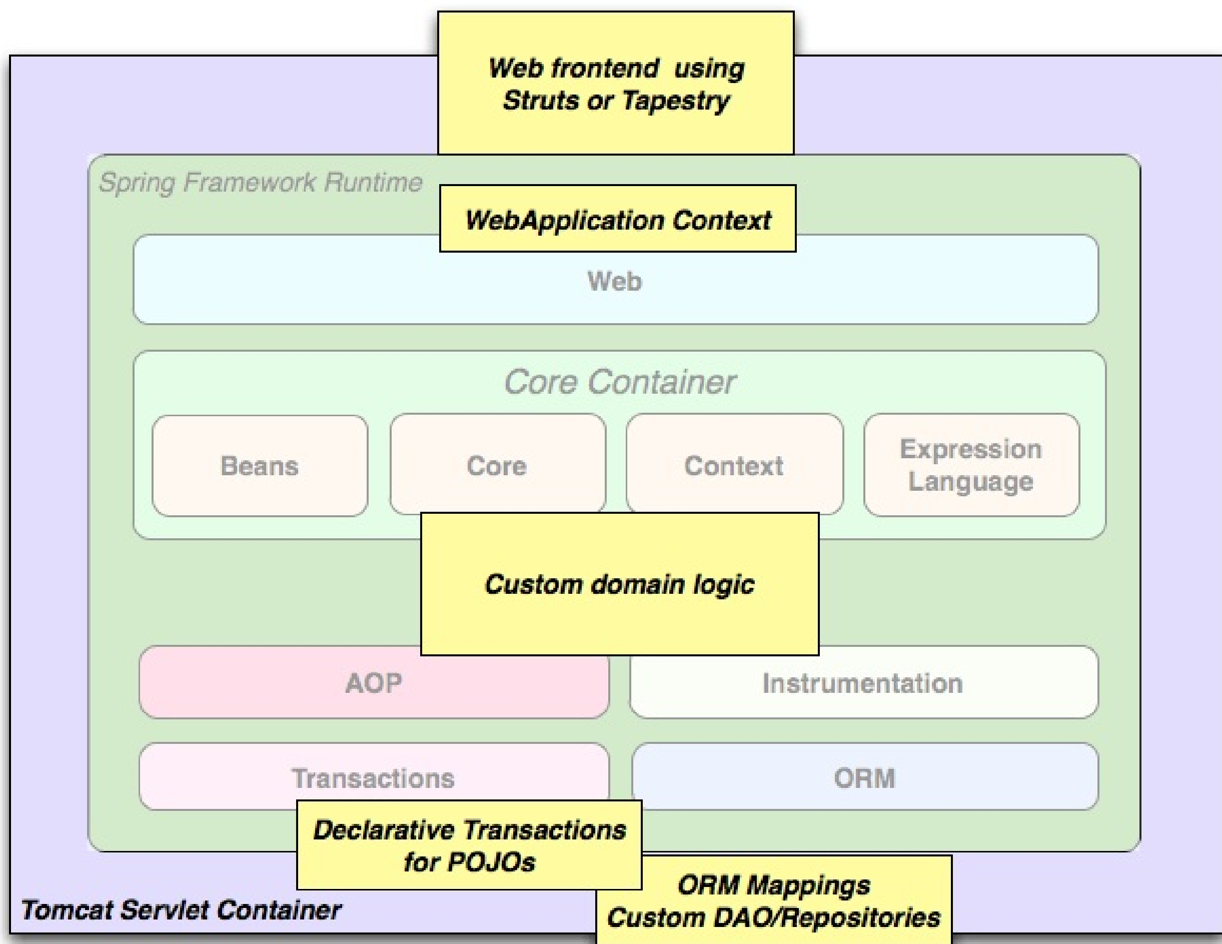
之前描述的构建模块使Spring成为许多应用场景的理性选择，从在资源受限设备上运行的嵌入式应用程序到使用Spring的事务管理功能和Web框架集成的全面的企业应用程序。

图2.2 典型的成熟完整的Spring Web应用程序



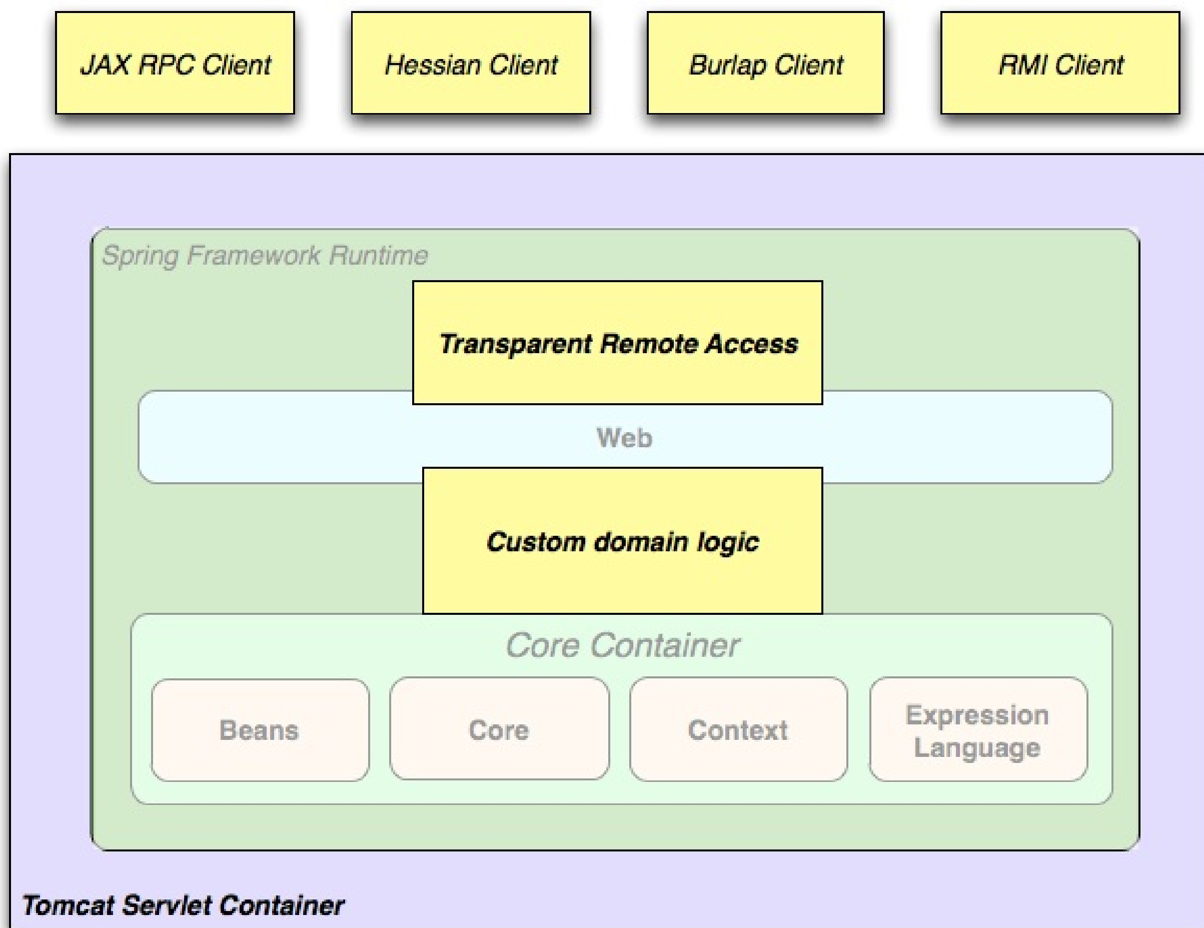
Spring的声明式事务管理功能使Web应用程序完全事务性，就像使用EJB容器管理的事务一样。所有您的定制业务逻辑都可以使用简单的POJO实现，并由Spring的IoC容器进行管理。附加服务包括支持发送电子邮件和独立于Web层的验证，可让您选择执行验证规则的位置。 Spring的ORM支持与JPA和Hibernate集成;例如，当使用Hibernate时，可以继续使用现有的映射文件和标准的Hibernate SessionFactory配置。表单控制器将Web层与域模型无缝集成，从而无需ActionForms或将HTTP参数转换为域模型的值的其他类。

图2.3 使用第三方web框架的Spring中间层



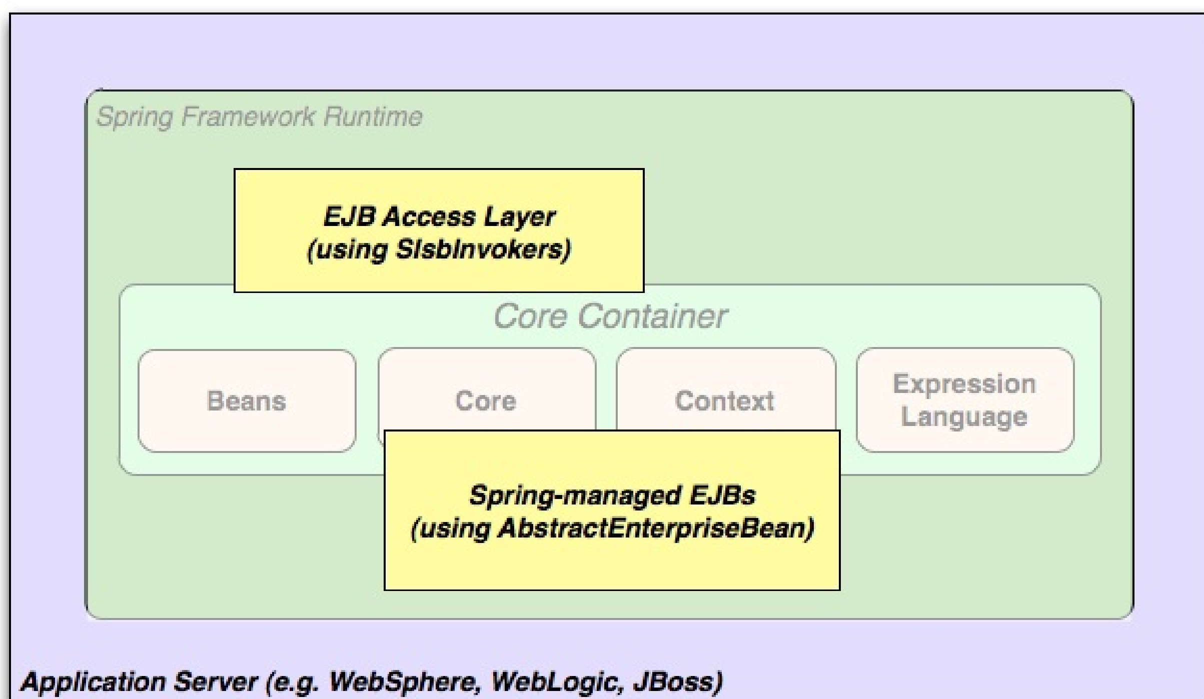
有时情况不允许你完全切换到不同的框架。 Spring框架并不强制您使用其中的一切;这不是一个全有或全无的解决方案。使用Struts，Tapestry，JSF或其他UI框架构建的现有前端可以与基于Spring的中间层集成，从而允许您使用Spring事务功能。您只需要使用ApplicationContext连接您的业务逻辑，并使用WebApplicationContext来集成您的Web层。

图*2.4* 远程使用场景



当您需要通过Web服务访问现有代码时，你可以使用Spring的 Hessian-，Rmi-或HttpInvokerProxyFactoryBean类。启用对现有应用程序的远程访问并不困难。

图*2.5 EJBs – *包装现有的*POJOs*



Spring Framework还为Enterprise JavaBeans提供了一个访问和抽象层，使您能够重用现有的POJO，并将其包装在无状态会话bean中，以便在可能需要声明式安全性的, 可扩展的，故障安全的Web应用程序中使用。

2.3.1依赖管理和命名约定

依赖关系管理和依赖注入是不同的。为了将Spring的这些不错的功能（如依赖注入）集成到应用程序中，您需要组装所有需要的库（jar文件），并在运行时导入到类路径（classpath）中，也有可能在编译时就需要加入类路径。这些依赖关系不是注入的虚拟组件，而是文件系统中的物理资源（通常是这样）。依赖关系管理的过程包括定位这些资源，存储它们并将其添加到类路径中。依赖关系可以是直接的（例如，我的应用程序在运行时依赖于Spring）或间接的（例如我的应用程序依赖于commons-dbcp ，而commons-dbcp 又依赖于commons-pool）。间接依赖关系具有“传递性”，它们是最难识别和管理的依赖关系。

如果你要使用Spring，你需要获得一个包含你所需要的Spring模块的jar库的副本。为了使这更容易，Spring被打包为一组尽可能分离依赖关系的模块，例如，如果您不想编写Web应用程序，则不需要spring-web模块。要引用本指南中的Spring库模块，我们使用一个简写命名约定spring- *或spring – *.jar，其中*表示模块的简称（例如spring-core，spring-webmvc，spring-jms等））。您实际使用的jar文件名通常是与版本号连接的模块名称（例如spring-core-5.0.0.M5.jar）。

Spring框架的每个版本都会发布到以下几个地方：

- Maven Central，它是Maven查询的默认存储库，不需要任何特殊配置。Spring的许多常见的库也可以从Maven Central获得，Spring社区的大部分使用Maven进行依赖关系管理，所以这对他们来说很方便。这里的jar的名字是spring – * – <version> .jar，Maven groupId是org.springframework。
- 在专门用于Spring的公共Maven存储库中。除了最终的GA版本，该存储库还承载开发快照和里程碑版本。 jar文件名与Maven Central格式相同，因此这是一个有用的地方，可以将开发中的版本的Spring与在Maven Central中部署的其他库配合使用。该存储库还包含集中分发的zip文件，其中包含所有Spring jar，捆绑在一起以便于下载。

所以你需要决定的第一件事是如何管理你的依赖关系：我们通常建议使用像Maven，Gradle或Ivy这样的自动化系统，但你也可以通过自己下载所有的jar来手动执行。

您将在下面找到Spring artifacts列表。有关每个模块的更完整的描述，第2.2节“模块”.

表2.1 Spring框架的Artifacts

GroupId	**ArtifactId**	**Description(描述)**
org.springframework	spring-aop	Proxy-based AOP support
org.springframework	spring-aspects	AspectJ based aspects
org.springframework	spring-beans	Beans support, including Groovy
org.springframework	spring-context	Application context runtime, including scheduling and remoting abstractions
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context
org.springframework	spring-core	Core utilities, used by many other Spring modules
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
org.springframework	spring-jms	JMS support package, including helper classes to send and receive JMS messages
org.springframework	spring-messaging	Support for messaging architectures and protocols
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support
org.springframework	spring-oxm	Object/XML Mapping
org.springframework	spring-test	Support for unit testing and integration testing Spring components

org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration
org.springframework	spring-web	Web support packages, including client and web remoting
org.springframework	spring-webmvc	REST Web Services and model-view-controller implementation for web applications
org.springframework	spring-websocket	WebSocket and SockJS implementations, including STOMP support

Spring的依赖和依赖于Spring

虽然Spring为大量企业和其他外部工具提供集成和支持，但它有意将其强制性的依赖保持在最低限度：在使用Spring用于简单的用例时，您不必定位和下载（甚至自动的去做）大量的jar库。对于基本依赖注入功能，只有一个强制性的外部依赖关系，也就是用于日志记录的依赖（有关日志记录选项的更详细描述，请参阅下文）。

接下来，我们概述了配置依赖于Spring的应用程序所需的基本步骤，首先是使用Maven，然后使用Gradle，最后使用Ivy。在任何情况下，如果不清楚，请参阅依赖关系管理系统的文档，或查看一些示例代码 – Spring本身在构建时使用Gradle来管理依赖关系，我们的示例主要使用Gradle或Maven。

Maven的依赖管理

如果您使用Maven进行依赖关系管理，则甚至不需要显式提供依赖关系。 例如，要创建应用程序上下文并使用依赖注入来配置应用程序，您的Maven依赖配置如下所示：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.0.M5</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

（依赖配置）就是这样。注意，如果您不需要针对Spring API进行编译，那么范围(scope)可以被声明为运行时，通常情况下这是基本的依赖注入用例。

以上示例适用于Maven Central存储库。要使用Spring Maven仓库（例如，使用里程碑或开发中的快照版本），您需要在Maven配置中指定仓库位置。完整版本：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>http://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

对于里程碑(milestones)：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

而对于快照（snapshots）：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

Maven的“材料清单”依赖

使用Maven时，可能会意外混合不同版本的Spring JAR。例如，您可能会发现第三方库或另一个Spring项目会传递依赖于旧版本的Spring JARs。如果您忘记自己明确声明直接依赖，可能会出现各种意外问题。

为了克服这些问题，Maven支持“材料清单(bill of materials)”（BOM）依赖的概念。您可以在dependencyManagement部分中导入spring-framework-bom，以确保所有Spring依赖（直接和传递）都是相同的版本

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>5.0.0.M5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

使用BOM的另外一个好处是，您不再需要在依赖于Spring Framework artifacts时指定<version>属性：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
<dependencies>
```

Gradle 依赖管理

要使用Gradle 构建系统的Spring仓库，请在仓库部分中包含适当的URL：

```
repositories {
    mavenCentral()
    // and optionally...
    maven { url "http://repo.spring.io/release" }
}
```

您可以根据需要将repositoriesURL从/release更改为/milestone或/snapshot。一旦repositories被配置，你可以通常使用Gradle方式来声明依赖：

```
dependencies {
    compile("org.springframework:spring-context:5.0.0.M5")
    testCompile("org.springframework:spring-test:5.0.0.M5")
}
```

Ivy依赖管理

如果您喜欢使用 Ivy 来管理依赖关系，那么还有类似的配置选项。 要配置Ivy指向Spring仓库，请将以下解析器添加到您的ivysettings.xml中：

```
<resolvers>
    <ibiblio name="io.spring.repo.maven.release"    m2compatible
="true" root="http://repo.spring.io/release/"/>
</resolvers>
```

您可以更改root从URL /release/到/milestone/或/snapshot/适当。

配置完成后，您可以在通常的方式添加依赖。例如（在ivy.xml）：

您可以根据需要将rootURL从 /release/更改为/milestone/或/snapshot/。 配置完成后，您可以按通常的方式添加依赖项。例如（在ivy.xml中）：

```
<dependency org="org.springframework"    name="spring-core" rev=
"5.0.0.M5" conf="compile->runtime"/>
```

Zip文件发行

虽然使用依赖关系管理的构建系统是推荐的获取Spring框架的方法，但仍然可以下载发布的zip文件。

Zip文件发布到Spring Maven存储库（这仅仅是为了我们的方便，您不需要使用Maven或任何其他构建系统才能下载它们）。

要下载发布的zip文件，打开Web浏览器到http://repo.spring.io/release/org/springframework/spring，并为所需的版本选择相应的子文件夹。zip文件以-dist.zip结尾，例如spring-framework- {spring-version} -RELEASE-dist.zip。里程碑和 快照也会发布在这里。

2.3.2 日志

日志是Spring非常重要的依赖，因为a）它是唯一的强制性外部依赖关系，b）每个人都喜欢看到他们使用的工具的一些输出，以及c）Spring集成了许多其他工具，都会具有日志依赖关系。应用程序开发人员的目标之一通常是将统一的日志配置放在整个应用程序的中央位置，包括所有外部组件。这比以前有更多的困难，因为有这么多的日志框架可以选择。

Spring中的强制性日志依赖关系是Jakarta Commons Logging API（JCL）。我们针对JCL进行编译，我们还使JCL Log 对象对于扩展了Spring Framework的类可见。对于用户来说，所有版本的Spring都使用相同的日志库很重要：迁移很简单，因为即使扩展了Spring的应用程序，但仍然保留了向后兼容性。我们这样做的方式是使Spring中的一个模块显式地依赖于commons-logging（遵循JCL规范的实现），然后在编译时使所有其他模块依赖于它。例如，如果您使用Maven，并且想知道在哪里可以获取对commons-logging的依赖，那么它来自Spring，特别是来自名为spring-core的中央模块。

commons-logging 的好处在于，您不需要任何其他操作来使您的应用程序正常工作。它具有运行时发现算法，可以在类路径中查找其他日志框架，并使用它认为合适的日志框架（或者您可以告诉它需要哪一个）。如果没有其他可用的，您可以从JDK（简称java.util.logging或JUL）获得好用的日志框架。您应该发现，在大多数情况下，您的Spring应用程序可以快乐地把日志输出到控制台，这很重要。

不使用Commons Logging

不幸的是，commons-logging中的运行时发现算法虽然对最终用户很方便，但是也有很多问题。如果我们可以把时空倒回，现在开始使用Spring去开始一个新的项目，我们会使用不同的日志依赖关系。第一选择可能是Simple Logging Facade for Java（SLF4J），它也被许多其他使用Spring的应用工具所使用。

基本上有两种方法来关闭commons-logging：

1. 排除spring-core模块的依赖关系（因为它是明确依赖于commons-logging的唯一模块）
2. 依赖于一个特殊的commons-logging依赖关系，用空的jar代替库（更多的细节可以在 SLF4J FAQ 中找到）

要排除commons-logging，请将以下内容添加到dependencyManagement部分：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.0.M5</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

现在这个应用程序可能已经被破坏了，因为在类路径中没有实现JCL API，所以要修复它，必须提供一个新的JCL API。在下一节中，我们向您展示如何使用SLF4J提供JCL的替代实现。

使用SLF4J

SLF4J是一个更清洁的依赖关系，在运行时比commons-logging更有效率，因为它使用编译时绑定，而不是其集成的其他日志框架的运行时发现。这也意味着你必须更加明确地说明你在运行时想要发生什么，并声明它或相应地进行配置。 SLF4J提供对许多常见的日志框架的绑定，因此通常可以选择一个已经使用的日志框架，并绑定到该框架进行配置和管理。

SLF4J提供了绑定到许多常见的日志框架的方法，包括JCL，它也是可以反转的：是其他日志框架和自身(Spring)之间的桥梁。所以要使用SLF4J与Spring，您需要使用SLF4J-JCL bridge替换commons-logging依赖关系。一旦完成，那么在Spring中日志调用将被转换为对SLF4J API的日志调用，因此如果应用程序中的其他库使用该API，那么您有一个统一的地方来配置和管理日志记录。

常见的选择可能是将Spring链接到SLF4J，然后提供从SLF4J到Log4j的显式绑定。您需要提供多个依赖关系（并排除现有的commons-logging）：the bridge，Log4j的SLF4J实现和Log4j实现本身。在Maven你会这样做：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.0.M5</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.7.22</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>2.7</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.7</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.7</version>
    </dependency>
</dependencies>
```

使用的Log4j

> Log4j的1.x版本已经寿终正寝，以下的内容特指Log4j 2

许多人使用Log4j 作为配置和管理日志的日志框架。它是高效和成熟的，当我们构建和测试Spring，实际上它是在运行时使用的。 Spring还提供了一些用于配置和初始化Log4j的实用功能，因此它在某些模块中对Log4j具有可选的编译时依赖性。

要使用JCL和Log4j，所有你需要做的就是把Log4j加到类路径，并为其提供一个配置文件（log4j2.xml，log4j2.properties或其他 支持的配置格式）。对于Maven用户，所需的最少依赖关系是：

```
<dependencies>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.7</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>2.7</version>
    </dependency>
</dependencies>
```

如果你也想使用SLF4J，还需要以下依赖关系：

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.7</version>
  </dependency>
</dependencies>
```

下面是一个例子log4j2.xml用来把日志输出到控制台：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %lo
gger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="org.springframework.beans.factory" level="DEBU
G"/>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

运行时容器和原生JCL

许多人在那些本身提供JCL实现的容器中运行他们的Spring应用程序。IBM WebSphere应用服务器（WAS）为例。这往往会导致问题，遗憾的是没有一劳永逸的解决方案; 在大多数情况下，简单地从您的应用程序排除commons-logging是不够的。

要清楚这一点：报告的问题通常不是JCL本身，甚至commons-logging：而是将commons-logging绑定到另一个框架（通常是Log4j）。这可能会失败，因为commons-logging更改了在一些容器中发现的旧版本（1.0）和大多数人现在使用的版本（1.1）之间执行运行时发现的方式。 Spring不使用JCL API的任何不寻常的部分，所以没有什么破坏，但是一旦Spring或您的应用程序尝试输出日志，您可以发现与Log4j的绑定不起作用

在这种情况下，使用WAS最简单的方法是反转类加载器层次结构（IBM将其称为"parent last"），以便应用程序控制JCL依赖关系，而不是容器。该选项并不总是开放的，但是在公共领域还有许多其他建议可供选择，您的里程(集成程度)可能因容器的确切版本和功能集而异。

3. IOC容器

3.1 Spring IoC容器和beans的介绍

本章涵盖了Spring框架实现控制反转（IoC）[1]的原理。IoC又叫依赖注入（DI）。它描述了对象的定义和依赖的一个过程，也就是说，依赖的对象通过构造参数、工厂方法参数或者属性注入，当对象实例化后依赖的对象才被创建，当创建bean后容器注入这些依赖对象。这个过程基本上是反向的，因此命名为控制反转（IoC），它通过直接使用构造类来控制实例化，或者定义它们之间的依赖关系，或者类似于服务定位模式的一种机制。

`org.springframework.beans` 和 `org.springframework.context` 是Spring框架中IoC容器的基础， `BeanFactory` 接口提供一种高级的配置机制能够管理任何类型的对象。 `ApplicationContext` 是 `BeanFactory` 的子接口。它能更容易集成Spring的AOP功能、消息资源处理（比如在国际化中使用）、事件发布和特定的上下文应用层比如在网站应用中的 `WebApplicationContext`。

总之， `BeanFactory` 提供了配置框架和基本方法， `ApplicationContext` 添加更多的企业特定的功能。 `ApplicationContext` 是 `BeanFactory` 的一个子接口，在本章它被专门用于Spring的IoC容器描述。更多关于使用 `BeanFactory` 替代 `ApplicationContext` 的信息请参考章节 3.16, “The BeanFactory”。

在Spring中，由Spring IoC容器管理的对象叫做beans。 bean就是由Spring IoC容器实例化、组装和以其他方式管理的对象。此外bean只是你应用中许多对象中的一个。Beans以及他们之间的依赖关系是通过容器配置元数据反映出来。

3.2容器概述

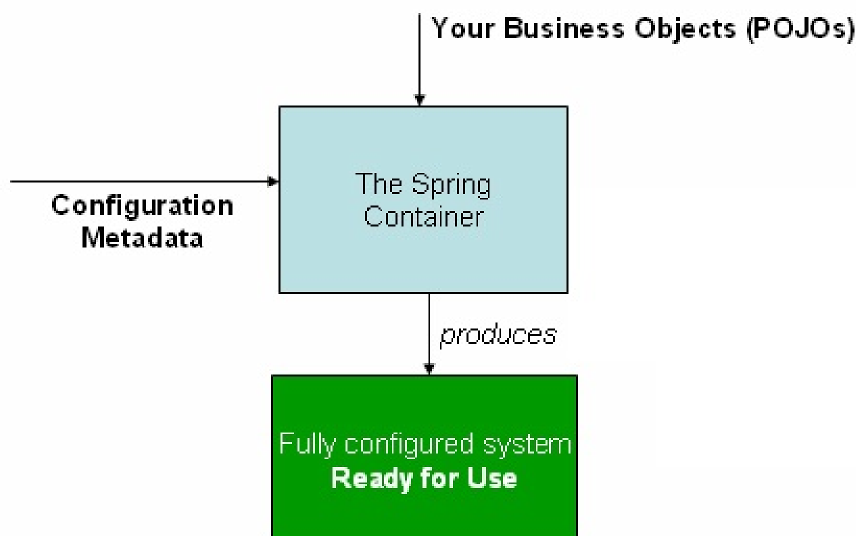
`org.springframework.context.ApplicationContext` 接口代表了Spring Ioc容器，它负责实例化、配置、组装之前的beans。容器通过读取配置元数据获取对象的实例化、配置和组装的描述信息。它配置的0元数据用xml、Java注解或Java代码表示。它允许你表示组成你应用的对象以及这些对象之间丰富的内部依赖关系。

Spring提供几个开箱即用的 `ApplicationContext` 接口的实现类。在独立应用程序中通常创建一个 `ClassPathXmlApplicationContext` 或 `FileSystemXmlApplicationContext` 实例对象。虽然XML是用于定义配置元数据的传统格式，你也可以指示容器使用Java注解或代码作为元数据格式，但要通过提供少量XML配置来声明启用对这些附加元数据格式的支持。

在大多数应用场景中，显示用户代码不需要实例化一个或多个Spring IoC容器的实例。比如在web应用场景中，在web.xml中简单的8行（或多点）样板式的xml配置文件就可以搞定（参见第3.15.4节"Web应用程序的便利的ApplicationContext实例化"）。如果你正在使用Eclipse开发环境中的Spring Tool Suite插件，你只需要鼠标点点或者键盘敲敲就能轻松搞定这几行配置。

下图是Spring如何工作的高级展示。你应用中所有的类都由元数据组装到一起所以当 `ApplicationContext` 创建和实例化后，你就有了一个完全可配置和可执行的系统或应用。

Figure 5.1. Spring IoC容器



3.2.1 配置元数据

如上图所示，Spring IoC容器使用了一种配置元数据的形式。此配置元数据表示应用程序的开发人员告诉Spring容器怎样去实例化、配置和装备你应用中的对象。

配置元数据传统上以简单直观的XML格式提供，本章大部分都使用这种格式来表达Spring IoC容器核心概念和特性。

> 基于XML的元数据不是允许配置元数据的唯一形式，Spring IoC容器与实际写入配置元数据的格式是分离的。这些天许多的开发者在他们的Spring应用中选择基于Java配置。

更多关于Spring容器使用其他形式的元数据信息，请查看：

- 基于注解配置： 在Spring2.5中有过介绍支持基于注解的配置元数据
- 基于Java配置： 从Spring3.0开始，由Spring JavaConfig提供的许多功能已经成为Spring框架中的核心部分。这样你可以使用Java程序而不是XML文件定义外部应用程序中的bean类。使用这些新功能，可以查看 `@Configuration` , `@Bean` , `@Import` 和 `@DependsOn` 这些注解

Spring配置由必须容器管理的一个或通常多个定义好的bean组成。基于XML配置的元数据中，这些bean通过标签定义在顶级标签内部。在Java配置中通常在使用 `@Configuration` 注解的类中使用 `@Bean` 注解方法。

这些bean的定义所对应的实际对象就组成了你的应用。通常你会定义服务层对象，数据访问层对象(DAO)，展现层对象比如Struts的 `Action` 实例，底层对象比如Hibernate的 `SessionFactories` ，JMS `Queues` 等等。通常在容器中不定义细粒度的域对象，因为一般是由DAO层或者业务逻辑处理层负责创建和加载这些域对象。但是，你可以使用Spring集成Aspectj来配置IoC容器管理之外所创建的对象。详情请查看Spring使用AspectJ依赖注入域对象

接下来这个例子展示了基于XML配置元数据的基本结构

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
        http://www.springframework.org/schema/beans/spring-beans
.xsd">

    <bean id="..." class="...">
        <!-- 在这里写 bean 的配置和相关引用 -->
    </bean>

    <bean id="..." class="...">
        <!-- 在这里写 bean 的配置和相关引用 -->
    </bean>

    <!-- 在这里配置更多的bean -->

</beans>
```

id属性用来使用标识每个独立的bean定义的字符串。 `class` 属性定义了bean的类型，这个类型必须使用全路径类名（必须是包路径+类名）。id属性值可以被依赖对象引用。该例中没有体现XML引用其他依赖对象。更多请查看bean的依赖。

3.12 基于Java的容器配置

3.12.1 基本概念：@Bean 和 @Configuration

最核心的是Spring支持全新的Java配置，例如@Configuration注解的类和@Bean注解的方法。

@Bean注解用来说明通过Spring IoC容器来管理时一个新对象的实例化，配置和初始化的方法。这对于熟悉Spring以XML配置的方式，@Bean和 element元素扮演了相同的角色。你可以在任何使用@Componen的地方使用@Bean，但是更常用的是在配置@Configuration的类中使用。

一个用@Configuration注解的类说明这个类的主要是作为一个bean定义的资源文件。进一步的讲，被@Configuration注解的类通过简单地在调用同一个类中其他的@Bean方法来定义bean之间的依赖关系。简单的@Configuration 配置类如下所示：

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

上面的AppConfig类和Spring XML 的配置是等价的：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"
/>
</beans>
```

> Full @Configuration vs 'lite' @Beans mode?
>
> 当@Bean方法在没有使用@Configuration注解的类中声明时，它们被称为以“lite”进行处理。例如，用@Component修饰的类或者简单的类中都被认为是“lite”模式。
>
> 不同于full @Configuration，lite @Bean 方法不能简单的在类内部定义依赖关系。通常，在“lite”模式下一个@Bean方法不应该调用其他的@Bean方法。
>
> 只有在@Configuration注解的类中使用@Bean方法是确保使用“full”模式的推荐方法。这也可以防止同样的@Bean方法被意外的调用很多次，并有助于减少在'lite'模式下难以被追踪的细小bug。

这个模块下我们深入的讨论了@Configuration和@Beans注解，首先我们将介绍基于Java配置的各种Spring容器的创建。

3.12.2 使用AnnotationConfigApplicationContext实例化Spring容器

下面的部分介绍Spring的AnnotationConfigApplicationContext，Spring 3.0的新内容。这个通用的ApplicationContext实现不仅可以接受@Configuration注解类为输入，还可以接受使用JSR-330元数据注解的简单类和@Component类。

当@Configuration注解的类作为输入时，@Configuration类本身会被注册为一个bean，在这个类中所有用@Bean注解的方法都会被定义为一个bean。

当使用@Component和JSR-330类时，它们被注册为bean的定义，并且假设在有必要时使用这些类内部诸如@Autowired或@Inject之类的DI元数据。

简单构造

实例化使用@Configuration类作为输入实例化AnnotationConfigApplicationContext和实例化ClassPathXmlApplicationContext时使用Spring的XML文件作为输入的方式大致相同。这在无XML配置的Spring容器时使用：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

如上所述，AnnotationConfigApplicationContext不限于仅使用@Configuration类。任何@Component或JSR-330注解的类都可以作为输入提供给构造函数。例如：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(MyServiceImpl.class, Dependency1.class, Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面假设MyServiceImpl、Dependency1和Dependency2都用了Spring的依赖注入的注解，例如@Autowired。

使用**register(Class<?>…)**的方式构建容器

也可以使用无参构造函数实例化AnnotationConfigApplicationContext，然后使用register()方法配置。当使用编程方式构建AnnotationConfigApplicationContext时，这种方法特别有用。

使用**scan**（**String …**）组件扫描

启用组件扫描，只需要在你的@Configuration类中做如下配置：

```
@Configuration
@ComponentScan(basePackages = "com.acme")
public class AppConfig  {
    ...
}
```

有Spring使用经验的用户，对Spring XML的context的声明非常熟悉：

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

在上面的例子中，com.acme将会被扫描，它会寻找任何@Component注解的类，这些类将会在Spring的容器中被注册成为一个bean。AnnotationConfigApplicationContext暴露的scan(String…)方法以达到相同组件扫描的功能：

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfi
gApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

记住：使用@Configuration注解的类是使用@Component进行元注解，所以它们也是组件扫描的候选，假设AppConfig被定义在com.acme这个包下（或者它下面的任何包），它们都会在调用scan()方法期间被找出来，然后在refresh()方法中它们所有的@Bean方法都会被处理，在容器中注册成为bean。

AnnotationConfigWebApplicationContext对于web应用的支持

AnnotationConfigApplicationContext在WebApplicationContext中的变体为AnnotationConfigWebApplicationContext。当配置Spring ContextLoaderListener servlet 监听器、Spring MVC DispatcherServlet的时候，可以用此实现。下面为配置典型的Spring MVC DispatcherServlet的web.xml代码段。注意contextClass上下文参数和init-param的使用：

```
<web-app>
    <!-- Configure ContextLoaderListener to use AnnotationConfig
WebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
```

```
            org.springframework.web.context.support.AnnotationCo
nfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more com
ma- or space-delimited
        fully-qualified @Configuration classes. Fully-qualified
packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using C
ontextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextL
oaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.Dispatche
rServlet</servlet-class>
        <!-- Configure DispatcherServlet to use AnnotationConfig
WebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.web.context.support.Annotati
onConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more
 comma- or space-delimited
            and fully-qualified @Configuration classes -->
```

```
            <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>com.acme.web.MvcConfig</param-value>
            </init-param>
        </servlet>

        <!-- map all requests for /app/* to the dispatcher servlet -
->
        <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>/app/*</url-pattern>
        </servlet-mapping>
</web-app>
```

3.12.3 使用@Bean注解

@Bean是XML元素方法级注解的直接模拟。它支持由提供的一些属性，例如：init-method，destroy-method，autowiring和name。

您可以在@Configuration或@Component注解的类中使用@Bean注解。

定义一个**bean**

要定义一个bean，只需在一个方法上使用@Bean注解。您可以使用此方法在指定方法返回值类型的ApplicationContext中注册bean定义。默认情况下，bean名称与方法名称相同。以下是@Bean方法声明的简单示例：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

这种配置完全和下面的Spring XML配置等价：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceIm
pl"/>
</beans>
```

两种声明都可以使得一个名为transferService的bean在ApplicationContext可用，绑定到TransferServiceImpl类型的对象实例上：

transferService -> com.acme.TransferServiceImpl

Bean 依赖

@Bean注解方法可以具有描述构建该bean所需依赖关系的任意数量的参数。例如，如果我们的TransferService需要一个AccountRepository，我们可以通过一个方法参数实现该依赖：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository acc
ountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}
```

这种解决原理和基于构造函数的依赖注入几乎相同，请参考相关章节，查看详细信息。

生命周期回调

任何使用了@Bean定义了的类都支持常规生命周期回调，并且可以使用JSR-250中的@PostConstruct和@PreDestroy注解，详细信息，参考JSR-250注解。

完全支持常规的Spring生命周期回调。如果一个bean实现了InitializingBean，DisposableBean或Lifecycle接口，它们的相关方法就会被容器调用。

完全支持*Aware系列的接口，例如：BeanFactoryAware，BeanNameAware，MessageSourceAware，ApplicationContextAware等。

@Bean注解支持任意的初始化和销毁回调方法，这与Spring XML 中bean元素上的init方法和destroy-method属性非常相似：

```
public class Foo {
    public void init() {
        // initialization logic
    }
}

public class Bar {
    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public Foo foo() {
        return new Foo();
    }

    @Bean(destroyMethod = "cleanup")
    public Bar bar() {
        return new Bar();
    }

}
```

默认情况下，使用Java config定义的具有公开关闭或停止方法的bean将自动加入销毁回调。如果你有一个公开的关闭或停止方法，但是你不希望在容器关闭时被调用，只需将@Bean（destroyMethod =""）添加到你的bean定义中即可禁用默认（推测）模式。 默认情况下，您可能希望通过JNDI获取资源，因为它的生命周期在应用程序之外进行管理。特别地，请确保始终为DataSource执行此操作，因为它已知在Java EE应用程序服务器上有问题。

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

另外，通过@Bean方法，通常会选择使用编程来进行JNDI查找：要么使用Spring的JndiTemplate/JndiLocatorDelegate帮助类，要么直接使用JNDI InitialContext，但不能使用JndiObjectFactoryBean变体来强制将返回类型声明为FactoryBean类型以代替目标的实际类型，它将使得在其他@Bean方法中更难用于交叉引用调用这些在此引用提供资源的方法。
当然上面的Foo例子中，在构造期间直接调用init（）方法同样有效：

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.init();
        return foo;
    }

    // ...

}
```

当您直接在Java中工作时，您可以对对象执行任何您喜欢的操作，并不总是需要依赖容器生命周期！

指定**bean**的作用域

使用**@Scope**注解

你可以指定@Bean注解定义的bean应具有的特定作用域。你可以使用Bean作用域章节中的任何标准作用域。

默认的作用域是单例，但是你可以用@Scope注解重写作用域。

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }

}
```

@Scope和**scope** 代理

Spring提供了一个通过范围代理来处理范围依赖的便捷方法。使用XML配置创建此类代理的最简单方法是元素。使用@Scope注解配置Java中的bean提供了与proxyMode属性相似的支持。默认是没有代理（ScopedProxyMode.NO），但您可以指定ScopedProxyMode.TARGET_CLASS或ScopedProxyMode.INTERFACES。

如果你使用Java将XML参考文档（请参阅上述链接）到范围的@Bean中移植范围限定的代理示例，则它将如下所示
如果你将XML 参考文档的scoped代理示例转化为Java @Bean，如下所示：

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
public UserPreferences userPreferences() {
    return new UserPreferences();
}

@Bean
public Service userService() {
    UserService service = new SimpleUserService();
    // a reference to the proxied userPreferences bean
    service.setUserPreferences(userPreferences());
    return service;
}
```

自定义**Bean**命名

默认情况下，配置类使用@Bean方法的名称作为生成的bean的名称。但是，可以使用name属性来重写此功能。

```
@Configuration
public class AppConfig {

    @Bean(name = "myFoo")
    public Foo foo() {
        return new Foo();
    }

}
```

Bean别名

如3.3.1节”bean 命名”中所讨论的，有时一个单一的bean需要给出多个名称，称为bean别名。 为了实现这个目标，@Bean注解的name属性接受一个String数组。

```
@Configuration
public class AppConfig {

    @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }

}
```

Bean描述

有时候需要提供一个详细的bean描述文本是非常有用的。当对bean暴露（可能通过JMX）进行监控使，特别有用。

可以使用@Description注解对Bean添加描述：

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Foo foo() {
        return new Foo();
    }

}
```

3.12.4 使用@Configuration注解

@Configuration是一个类级别的注解，指明此对象是bean定义的源。@Configuration类通过public @Bean注解的方法来声明bean。在@Configuration类上对@Bean方法的调用也可以用于定义bean之间的依赖。概述，请参考第3.12.1节”基本概念：@Bean和@Configuration”。

bean依赖注入

当@Beans相互依赖时，表示依赖关系就像一个bean方法调用另一个方法一样简单：

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }

}
```

上面的例子，foo接受一个bar的引用来进行构造器注入：

> 这种方法声明的bean的依赖关系只有在@Configuration类的@Bean方法中有效。你不能在@Component类中来声明bean的依赖关系。

方法查找注入

如前所述，方法查找注入是一个你很少用用到的高级特性。在单例的bean对原型的bean有依赖性的情况下，它非常有用。这种类型的配置使用，Java提供了实现此模式的自然方法。

```
public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interf
ace
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command in
stance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

使用Java支持配置，您可以创建一个CommandManager的子类，覆盖它抽象的createCommand()方法，以便它查找一个新的（原型）命令对象：

```
@Bean
@Scope("prototype")
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager wit
h command() overridden
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}
```

有关基于**Java**配置内部如何工作的更多信息

下面的例子展示了一个@Bean注解的方法被调用两次：

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl(
);
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl(
);
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }

}
```

clientDao()方法clientService1()和clientService2()被各自调用了一次。因为这个方法创建并返回了一个新的ClientDaoImpl实例，你通常期望会有2个实例（每个服务各一个）。这有一个明显的问题：在Spring中，bean实例默认情况下是单例。神奇的地方在于：所有的@Configuration类在启动时都使用CGLIB进行子类实例化。在子类中，子方法在调用父方法创建一个新的实例之前会首先检查任何缓存(作用域)的bean。注意，从Spring 3.2开始，不再需要将CGLIB添加到类路径中，因为CGLIB类已经被打包在org.springframework.cglib下，直接包含在spring-core JAR中。

> 根据不同的bean作用域，它们的行为也是不同的。我们这里讨论的都是单例模式。
>
> 这里有一些限制是由于CGLIB在启动时动态添加的特性，特别是配置类都不能是final类型。然而从4.3开始，配置类中允许使用任何构造函数，包含@Autowired使用或单个非默认构造函数声明进行默认注入。如果你希望避免CGLIB带来的任何限制，那么可以考虑子在非@Configuration注解类中声明@Bean注解方法。例如，使用@Component注解类。在 @Bean方法之间的交叉调用不会被拦截，所以你需要在构造器或者方法级别上排除依赖注入。

3.12.5 基于Java组合配置

使用**@Import**注解

和Spring XML文件中使用元素来帮助模块化配置类似，@Import注解允许从另一个配置类加载@Bean定义：

```
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }

}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }

}
```

现在，在实例化上下文时不是同时指明ConfigA.class和ConfigB.class，而是仅仅需要明确提供ConfigB：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(ConfigB.class);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

这种方法简化了容器实例化，因为只需要处理一个类，而不是需要开发人员在构建期间记住大量的@Configuration注解类。

> 从Spring Framework 4.2开始，@Import注解也支持对常规组件类的引用，类似AnnotationConfigApplicationContext.register方法。如果你希望避免组件扫描，使用一些配置类作为所有组件定义的入口，这个方法特别有用。

引入的**@Bean**定义中注入依赖

上面的类中可以运行，但是太过简单。在大多数实际场景中，bean在配置类之间相互依赖。当使用XML时，这没有问题，因为没有编译器参与，一个bean可以简单的声明为ref="someBean"并且相信Spring在容器初始化过程处理它。当然，当使用@Configuration注解类，Java编译器会对配置模型放置约束，以便其他对其他引用的bean进行Java语法校验。
幸运的是，解决这个问题也很简单。正如我们讨论过的，@Bean方法可以有任意数量的参数来描述bean的依赖。让我们考虑一下真实场景和一系列@Configuration类，每个bean都依赖了其他配置中声明的bean：

```
@Configuration
public class ServiceConfig {

    @Bean
    public TransferService transferService(AccountRepository acc
ountRepository) {
        return new TransferServiceImpl(accountRepository);
    }

```

```
}

@Configuration
public class RepositoryConfig {

    @Bean
    public AccountRepository accountRepository(DataSource dataSo
urce) {
        return new JdbcAccountRepository(dataSource);
    }

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

这里有其他的方法实现相同的结果。记住@Configuration类最终只是容器中的另一个bean：这意味着它们可以像任何其他bean一样利用@Autowired和@Value注入等！

> 确保您注入的依赖关系是最简单的。@Configuration类在上下文初始化期间处理，强制要求依赖使用这种方式进行注入可能导致意外的早期初始化问题。如果可能，就采用如上述例子所示的基于参数的注入。
>
> 同时，也要特别小心通过@Bean的BeanPostProcessor BeanFactoryPostProcessor定义。它们应该被声明为static的@Bean方法，不会触发包含它们的配置类的实例化。否则，@Autowired和@Value将在配置类上不生效，因为它太早被创建为一个实例了。

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    @Autowired
    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

}
```

```
@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

> 只有Spring Framework 4.3才支持@Configuration类中的构造方法注入。注意，如果目标bean只定义了一个构造函数，那么则不需要指定@Autowired；如果目标bean只定义一个构造函数，则不需要指定@Autowired;在上面的例子中，@Autowired在RepositoryConfig构造函数中是不必要的。
>
> 在上面的场景中，使用@Autowired可以很好的提供所需的模块化，但是准确的决定在哪里自动注入定义的bean还是模糊的。例如，作为开发者来看待ServiceConfig，如何准确的确定自动注入 AccountRepository 是在哪里声明的？它没有明确的出现在代码中，这可能还不错。记住，Spring Tool Suite 提供工具可以渲染图形展示对象是如何装配的，这些可能是你所需要的。同时，你的Java IDE 也可以很简单的找出所有AccountRepository类型的声明和使用，这将很快的展示出你@Bean方法的位置和返回类型。

在这种歧义不可接受的情况下，你希望从IDE中直接从一个@Configuration类导航到另一个类，可以考虑自动装配配置类的本身：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean meth
od!
        return new TransferServiceImpl(repositoryConfig.accountR
epository());
    }

}
```

在上面的情形中，定义的AccountRepository是完全透明的。但是，ServiceConfig和RepositoryConfig是紧密耦合在一起了；这需要权衡。这种紧密耦合的可以通过基于接口或者抽象@Configuration类来缓解。可以考虑下面的代码：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountR
epository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();
```

```
}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig
 {

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }

}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) //
 import the concrete config!
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(SystemTestConfig.class);
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

现在ServiceConfig与具体DefaultRepositoryConfig就是送耦合，IDE内置的工具也仍然有用：对开发人员来说，可以轻松的获取RepositoryConfig实现层级类型。用这种方法，定位到@Configuration类及它的依赖类和定位基于接口的代码就没什么区别。

有条件的包括**@Configuration**类或**@Bean**方法

通常，有条件的开启或者禁用一个完整的@Configuration类，甚至是基于有任意系统状态的单独@Bean方法。一个常见的例子就是使用@Profile注解来激活仅在Spring 环境中启用的特定的profile文件（有关详细信息，请参阅第3.13.1节“Bean definition profiles”）
@Profile注解是用一个更加灵活的@Conditional注解实现的。@Conditional注解表示@Bean在被注册前应该查阅特定的
org.springframework.context.annotation.Condition实现。
Condition接口的实现只提供了一个返回true或者false的matches(...)方法。例如
@Profile是Condition的具体实现：

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMe
tadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs = metadata.getAllAnn
otationAttributes(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((S
tring[]) value))) {
                    return true;
                }
            }
            return false;
        }
    }
    return true;
}
```

@Conditional详细信息请参考javadocs。

Java and XML 混合配置

Spring 对@Configuration配置类的支持的目的不是100%来替换Spring XML配置的。一些基本特性，例如：Spring XML命名空间仍然是容器配置的一个理想方式。在 XML更便于使用或者是必须使用的情况下，要么以“XML为中心”的方式来实例化

容器，比如，ClassPathXmlApplicationContext，要么以“Java为中心”的方式，使用AnnotationConfigurationApplicationContext 和@ImportResource注解来引入所需的XML。

以XML为中心使用@Configuration类

假设你可能会以XML包含@Configuration类的方式来启动一个Spring容器。例如，在一个现有使用Spring XML的大型代码库中，根据需要从已有的XML文件中创建@Configuration类是很简单的。下面你可以发现在以XML为中心的情形下使用@Configuration类的选项。谨记，@Configuration类最终只是容器中的一个bean。在这个例子中，我们会创建一个名为AppConfig的@Configuration类，它作为一个bean的定义包含在system-test-config.xml中。因为context:annotation-config/是打开的，容器会识别@Configuration，并且会处理AppConfig中声明的@Bean方法。

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }

}
```

system-test-config.xml:

```
<beans>
    <!-- enable processing of annotations such as @Autowired and
 @Configuration -->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/
jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <bean class="org.springframework.jdbc.datasource.DriverManag
erDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

jdbc.properties:

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

```
public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
"classpath:/com/acme/system-test-config.xml");
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    // ...
}
```

> 在上面的system-test-config.xml中，AppConfig中的bean没有声明id的元素。然而它也可以被接受，在没有其他bean引用的情况下也没有必要给出，也不太可能通过名字的方式从容器中显式取出。像DataSource bean一样，只能通过类型自动注入，所以明确的bean id也不严格要求。

因为@Configuration是@Component的一个元注解，对于component的扫描@Configuration注解类会自动成为候选者。和上面的场景相同，利用component扫描可以重新定义system-test-config.xml。注意在个案例中，我们不需要明确的声明context:annotation-config/，因为开启context:component-scan/，功能是相同的。

system-test-config.xml:

```
<beans>
    <!-- picks up and registers AppConfig as a bean definition -
->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder location="classpath:/com/acme/
jdbc.properties"/>

    <bean class="org.springframework.jdbc.datasource.DriverManag
erDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

@Configuration类为中心的 XML @ImportResource的使用

在以@Configuration类为主要机制的配置容器的应用程序中，仍然有必要使用一些XML。在这些场景中，只需使用@ImportResource，并根据需要定义一些XML。这样实现了“以Java为中心”方式来配置容器，并将XML保持在最低限度。

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, passwo
rd);
    }

}
```

properties-config.xml

```
<beans>
    <context:property-placeholder location="classpath:/com/acme/
jdbc.properties"/>
</beans>
```

jdbc.properties

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(AppConfig.class);
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    // ...
}
```

3.13 环境抽象

在应用环境中，集成在容器的抽象环境模型有两个方面：profiles和properties。只有给出的profile被激活，一组逻辑命名的bean定义才会在容器中注册。无论是在XML中或者通过注解，bean都会被分配给一个profile。环境变量对象角色和profiles的关系来决定哪个profiles(如果有)处于当前激活状态，哪个profiles默认被激活。几乎在所有的应用中，Properties都扮演了一个重要的对象，这可能有各种来源：属性文件， JVM 系统属性文件，系统环境变量，JNDI，servlet上下文参数，属性查询对象，Maps等等。环境变量对象的角色和properties的关系用于配置属性并从中解析属性提供给用户一个便捷的服务接口。

3.13.1 Bean的profiles定义

Bean定义profiles是在核心容器中允许不同的bean在不同环境注册的机制。环境对于不同的用户意味着不同的东西，这个特性可以帮助许多用例，包括：

- 在开发中不使用内存中的数据源 VS 在质量测试或生产环境中从JNDI查找相同的数据源。
- 当把应用部署在可执行的环境中时注册监控基础架构
- 对于客户A注册的自定义实现VS.客户B部署

让我首先考虑在一个需要数据源的应用中使用这个例子。在测试环境中，配置可能如下：

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

让我现在考虑一下，如何把这个应用部署在测试环境或者生产环境中，假设应用所需的数据源将会被注册在生产应用环境中的JNDI目录。现在我们的数据源bean看起来像这样：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasourc
e");
}
```

问题就是如何根据当前的环境在这两种变量之间进行切换。随着时间的推移，Spring的用户已经设计了很多种方法来实现此功能，通常依赖于系统环境变量和包含${placeholder}的XML 语句，根据环境变量的值可以解决正确的文件路径配置。Bean定义profiles是容器为了解决这个问题而提供的一个核心功能。

如果我们概括一下上面bean定义环境变量的示例，我们最终需要在特定的上下文中注册特定的bean，而不是其他的。你可以说你想要在情形A中注册一个特定的Bean定义的profile，在情形B中是另外一个。我们首先看下如何更新我们的配置以反映这种需求。

@Profile

当一个或者多个特定的profiles被激活，@Profile注解允许你指定一个有资格的组件来注册。使用我们上面的例子，我们可以按照下面的重写dataSource配置：

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql
")
            .addScript("classpath:com/bank/config/sql/test-data.
sql")
            .build();
    }
}
```

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datas
ource");
    }
}
```

> 如前所述，使用@Bean方法，通常会选择使用程序话的JNDI查找：要么使用Spring的 JndiTemplate/JndiLocatorDelegate帮助要么直接使用上面展示的JNDI InitialContext，而不是强制声明返回类型为FactoryBean的JndiObjectFactoryBean变体。

@Profile可以被用作为创建一个自定义组合注解的元注解。下面的例子定义了一个@Production注解，它可以被用作替换@Profile（“production”）的注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

在仅仅包含一个特殊bean的配置类中，@Profile也可以被声明在方法级别：

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql
")
            .addScript("classpath:com/bank/config/sql/test-data.
sql")
            .build();
    }

    @Bean
    @Profile("production")
    public DataSource productionDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datas
ource");
    }
}
```

> 如果一个@Configuration类被标记为@Profile，那么所有的@Bean方法和@Import注解相关的类都会被忽略，除非一个或多个特别的profiles被激活。如果一个@Component或@Configuration类被标记为@Profile({“p1”, “p2”})，那么这个类将不会被注册/处理，除非被标记为’p1’和/或’p2’的profiles已经被激活。如果给出的profile的前缀带有取反的操作符(!)，那么注解的元素将会被注册，除非这个profile没有被激活。例如，给出@Profile({“p1”, “!p2”})，如果profile ‘p1’是激活状态或者profile ‘p2’不是激活状态的时候才会注册。

3.13.2 XML bean 定义 profiles

XML对应元素的profile属性。我们上面的示例配置可以重写为下面的两个XML配置：

```
<beans profile="dev"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/sch
ema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/tes
t-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jd
bc/datasource"/>
</beans>
```

也可以避免在同一个文件中分割和嵌套元素：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql
/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql
/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/en
v/jdbc/datasource"/>
    </beans>
</beans>
```

spring-bean.xsd 约束允许这样的元素仅作为文件中的最后一个元素。这有助于XML的灵活性，且不会产生混乱。

激活**profile**

现在我们已经更新了我们的配置，我们仍然需要说明哪个profile是激活的。如果我们现在开始我们示例应用程序，我们将会看到一个NoSuchBeanDefinitionException被抛出，因为容器找不到一个名为dataSource的Spring bean。
激活一个profile可以通过多种方式完成，但是大多数情况下，最直接的办法就是通过存在ApplicationContext当中的环境变量的API进行编程：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApp
licationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiD
ataConfig.class);
ctx.refresh();
```

除此之外，profiles也可以通过声明spring.profiles.active属性来激活，这个可以通过在系统环境变量，JVM系统属性，web.xml中的servlet上下文环境参数，甚至JNDI的入口（请参考 3.13.3, “PropertySource abstraction”）。在集成测试中，激活profiles可以通过在Spring-test模块中的@ActiveProfiles注解来声明（参见“使用profiles来配置上下文环境”章节）。

注意，profiles不是“二者选一”的命题；它可以一次激活多个profiles。以编程的方式来看，简单的传递多个profile名字给接受String 可变变量参数的setActiveProfiles()方法：

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

在声明式中，spring.profiles.active可以接受以逗号分隔的profile 名称列表：

```
-Dspring.profiles.active="profile1,profile2"
```

默认的**profile**

默认配置文件表示默认启用的配置文件。考虑以下几点：

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql
")
            .build();
    }
}
```

如果没有profile是激活状态，上面的dataSource将会被创建；这种方式可以被看做是对一个或者多个bean提供了一种默认的定义方式。如果启用任何的profile，那么默认的profile都不会被应用。
在上下文环境可以使用setDefaultProfiles()或者spring.profiles.default属性来修改默认的profile名字。

3.13.3 属性源抽象

Spring 环境抽象提供了可配置的属性源层次结构的搜索操作。为了充分的解释，请考虑下面的例子：

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsFoo = env.containsProperty("foo");
System.out.println("Does my environment contain the 'foo' proper
ty? " + containsFoo);
```

在上面的代码段中，我们看到了一个高级别的方法来要求Spring是否为当前环境定义foo属性。为了回答这个问题，环境对象对一组PropertySource对象执行搜索。一个PropertySource是对任何key-value资源的简单抽象，并且Spring 的标准环境是由两个PropertySource配置的，一个表示一系列的JVM 系统属性(System.getProperties()),一个表示一系列的系统环境变量(System.getenv())。

> 这些默认的属性资源存在于StandardEnvironment，可以在应用中独立使用。StandardServletEnvironment包含其他默认的属性资源，包括servlet配置和servlet上下文参数。它可以选择性的启用JndiPropertySource。详细信息请查看javadocs。

具体的说，当使用StandardEnvironment时，如果在运行时系统属性或者环境变量中包括foo，那么调用env.containsProperty(“foo”)方法将会返回true。

搜索是按照层级执行的。默认情况，系统属性优先于环境变量，所以这两个地方同时存在属性foo的时候，调用env.getProperty(“foo”)将会返回系统属性中的foo值。注意，属性值不会被合并而是被之前的值覆盖。对于一个普通的StandardServletEnvironment，它完整的层次结构如下，最顶端的优先级最高：

- ServletConfig参数(如果适用，例如DispatcherServlet上下文环境)
- ServletContext参数(web.xml中的context-param)
- JNDI环境变量(“java:comp/env/”)
- JVM系统属性(“-D”命令行参数)
- JVM系统环境变量(操作系统环境变量)

更重要的是，整个机制都是可配置的。也许你有个自定义的属性来源，你想把它集成到这个搜到里面。这也没问题，只需简单的实现和实例化自己的PropertySource，并把它添加到当前环境的PropertySources集合中：

```
ConfigurableApplicationContext ctx = new GenericApplicationConte
xt();
MutablePropertySources sources = ctx.getEnvironment().getPropert
ySources();
sources.addFirst(new MyPropertySource());
```

在上面的代码中，MyPropertySource被添加到搜索中的最高优先级。如果它包含了一个foo属性，在任何其他的PropertySource中的foo属性之前它会被检测到并返回。MutablePropertySources API暴露了很多允许精确操作该属性源集合的方法。

3.13.4 @PropertySource

@PropertySource注解对添加一个PropertySource到Spring的环境变量中提供了一个便捷的和声明式的机制。
给出一个名为”app.properties”的文件，它含了testbean.name=myTestBean的键值

对，下面的@Configuration类使用@PropertySource的方式来调用testBean.getName()，将会返回”myTestBean”。

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {
 @Autowired
 Environment env;

 @Bean
 public TestBean testBean() {
  TestBean testBean = new TestBean();
  testBean.setName(env.getProperty("testbean.name"));
  return testBean;
 }
}
```

任何出现在@PropertySource中的资源位置占位符都会被注册在环境变量中的资源解析。

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/a
pp.properties")
public class AppConfig {
 @Autowired
 Environment env;

 @Bean
 public TestBean testBean() {
  TestBean testBean = new TestBean();
  testBean.setName(env.getProperty("testbean.name"));
  return testBean;
 }
}
```

假设”my.placeholder”已经在其中的一个资源中被注册，例如：系统属性或环境变量，占位符将会被正确的值解析。如果没有，”default/path”将会使用默认值。如果没有默认值，而且无法解释属性，则抛出IllegalArgumentException异常。

3.13.5 声明中的占位符解决方案

以前，元素中占位符的值只能被JVM系统熟悉或者环境变量解析。现在已经解决了这种情况。因为抽象的环境已经通过容器被集成了，很容易通过它来分配占位符。这意味着你可以使用任何你喜欢的方式配置：可以通过系统属性和环境变量来改变搜索优先级，或者完全删除它们；可以以适当的方式添加你自己混合属性资源。

具体来说，下面的声明无论customer属性被定义在哪里，只要它存在环境变量中就有作用：

3.14 注册一个加载时编织器

在类被加载进JVM时Spring使用LoadTimeWeaver进行动态转换。
为了使的load-time weaving可用，那么你只需在配置了@Configuration的类上添加@EnableLoadTimeWeaving。

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {

}
```

相应的xml配置使用context:load-time-weaver元素：

```
<beans>
    <context:load-time-weaver/>
</beans>
```

一旦配置了ApplicationContext，那么在ApplicationContext中的任何bean都可以实现LoadTimeWeaverAware，从而接受对类加载时编织器实例的引用。这与Spring JPA支持相结合时非常有用，JPA类转化必须使用加载时编织。可以通过javadocs的LocalContainerEntityManagerFactoryBean获取更多详细信息，对于AspectJ加载时的编织请参考：

Section 7.8.4, “Load-time weaving with AspectJ in the Spring Framework”.

3.15 ApplicationContext的额外功能

正如本章开头所讨论的那样，org.springframework.beans.factory包提供基本的功能来管理和操作bean，包括以编程的方式。The org.springframework.context包增加了ApplicationContext接口，它继承了BeanFactory接口，除了以面向应用框架的风格扩展接口来提供一些额外的功能。很多人以完全声明的方式使用ApplicationContext，甚至没有以编程的方式去创建它，而是依赖诸如ContextLoader等支持类来自动的实例化ApplicationContext，作为Java EE web应用程序正常启动的一部分。
为了增强BeanFactory在面向框架风格的功能，上下文的包还提供了以下的功能：

- 通过MessageSource接口访问i18n风格的消息
- 通过ResourceLoader接口访问类似URL和文件资源
- 通过ApplicationEventPublisher接口，即bean实现ApplicationListener接口来进行事件发布
- 通过HierarchicalBeanFactory接口实现加载多个(分层)上下文，允许每个上下文只关注特定的层，例如应用中的web层

3.15.1 使用MessageSource 国际化

ApplicationContext接口继承了一个叫做MessageSource的接口，因此它也提供了国际化(i18n)的功能。Spring也提供了HierarchicalMessageSource接口，它可以分层去解析信息。这些接口共同为Spring消息效应解析提供了基础。这些接口上定义的方法包括：

- String getMessage(String code, Object[] args, String default, Locale loc): 这个基础的方法用来从MessageSource检索消息。当指定的区域中没有发现消息时，将使用默认的。任何参数传递都将使用标准库提供的MessageFormat变成替换值。
- String getMessage(String code, Object[] args, Locale loc): 本质上和前面提供的方法相同，只有一个区别，就是当没有指定消息，又没有发现消息，将会抛出NoSuchMessageException 异常。
- String getMessage(MessageSourceResolvable resolvable, Locale locale): 所有的属性处理方法都被包装在一个名为MessageSourceResolvable的类中，你可以使用此方法。

当ApplicationContext被载入的时候，它会自动的在上下文中去搜索定义的MessageSource bean。这个bean必须有messageSource的名称。如果找到这么一个bean，所有上述方法的调用都会委托给消息源。如果没有发现消息源，ApplicationContext会尝试寻找一个同名的父消息源。如果是这样，它会将那个bean作为MessageSource。如果ApplicationContext没有找到任何的消息源，那么一个空的DelegatingMessageSource将被实例化，以便能够接受到对上述定义方法的调用。

Spring提供了ResourceBundleMessageSource和StaticMessageSource两个MessageSource实现。它们两个都实现了HierarchicalMessageSource以便处理嵌套消息。StaticMessageSource很少使用，但是它提供了通过编程的方式增加消息源。下面展示ResourceBundleMessageSource使用的例子：

```
<beans>
    <bean id="messageSource"
            class="org.springframework.context.support.ResourceB
undleMessageSource">
        <property name="basenames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

在上面的例子中，假设在类路径下定义了format，exceptions和windows三个资源包。解析消息的任何请求都会通过ResourceBundles被JDK以标准方式处理。为了举例说明，假设上述两个资源包的文件内容是…

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

下面的实例展示了执行MessageSource功能的程序。记住所有的ApplicationContext的实现也是MessageSource的实现，而且它可以被强转为MessageSource接口。

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext
("beans.xml");
    String message = resources.getMessage("message", null, "Defa
ult", null);
    System.out.println(message);
}
```

上面的程序输出的结果为：
Alligators rock!

所以总结一下，MessageSource是定义在一个名为beans.xml，它存在类路径的跟目录下。messageSource bean定义通过basenames属性引用了很多的资源。在列表中传递给basenames属性的三个文件作为类路径下根目录中的文件存在，分别为format.properties, exceptions.properties, and windows.properties。

下一个例子展示传递给消息查找的参数；这些参数将会被转换为字符串并插入到消息查找的占位符中。

```
<beans>

    <!-- this MessageSource is being used in a web application -
->
    <bean id="messageSource" class="org.springframework.context.
support.ResourceBundleMessageSource">
        <property name="basename" value="exceptions"/>
    </bean>

    <!-- lets inject the above MessageSource into this POJO -->
    <bean id="example" class="com.foo.Example">
        <property name="messages" ref="messageSource"/>
    </bean>

</beans>
```

```
public class Example {

    private MessageSource messages;

    public void setMessages(MessageSource messages) {
        this.messages = messages;
    }

    public void execute() {
        String message = this.messages.getMessage("argument.requ
ired",
            new Object [] {"userDao"}, "Required", null);
        System.out.println(message);
    }

}
```

调用execute()方法的输出结果为:
The userDao argument is required.
关于国际化(i18n)，Spring的各种MessageSource实现遵循与标准JDK ResourceBundle相同的语言环境和回退规则。简而言之，并继续用前面messageSource 为例，如果你想根据英国(en-GB)解析消息，你可以创建这些文件format_en_GB.properties，exceptions_en_GB.properties, and windows_en_GB.properties。
通常，地域设置通过应用周围环境管理的。在此示例中，手动指定对(英国)区域消息进行解析。

in exceptions_en_GB.properties

argument.required=Ebagum lad, the {0} argument is required, I say, required.

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext
("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);
    System.out.println(message);
}
```

上面的程序运行输出为：

```
Ebagum lad, the ‘userDao’ argument is required, I say, required.
```

你也可以使用MessageSourceAware接口来获取对已定义MessageSource的引用。任何在ApplicationContext定义的bean都会实现MessageSourceAware，当bean被创建或者配置的时候，它会在应用上下文的MessageSource中被被注入。

> 作为ResourceBundleMessageSource的替代方法，Spring提供了一个ReloadableResourceBundleMessageSource类。这个变体支持同样打包文件格式，但是它更灵活而不是标准JDK基于ResourceBundleMessageSource的实现。特别的，它允许从任何Spring 资源位置读取文件(不仅仅是从类路径)而且还支持属性文件热加载(同时高效缓存他们)。ReloadableResourceBundleMessageSource的详细信息参考javadocs。

3.15.2 标准和自定义事件

ApplicationEvent类和ApplicationListener接口提供了ApplicationContext中的事件处理。如果一个bean实现了ApplicationListener接口，然后它被部署到上下问中，那么每次ApplicationEvent发布到ApplicationContext中时，bean都会收到通知。本质上，这是观察者模型。

> 从Spring 4.2开始，事件的基础得到了重要的提升，并提供了基于注解模型及任意事件发布的能力，这个对象不一定非要继承ApplicationEvent。当这个对象被发布时，我们把他包装在事件中。

Spring提供了一下的标准事件：

表**3.7** 内置事件

事件	解释
ContextRefreshedEvent	当ApplicationContext被初始化或者被刷新的时候发布，例如，在ConfigurableApplicationContext接口上调用refresh()方法。”初始化”在这里意味着所有的bean被加载，后置处理器被检测到并且被激活，单例的预加载，以及ApplicationContext对象可以使用。只要上下文还没有被关闭，refresh就可以被触发多次，前提所选的ApplicationContext支持热刷新。例如，XmlWebApplicationContext支持热刷新，而GenericApplicationContext不支持。
ContextStartedEvent	当ApplicationContext启动时发布，在ConfigurableApplicationContext接口上调用start()方法。”已启动”意味着所有bean的生命周期会接受到一个明确的启动信号。通常这个信号用来停止后的重启，但是他也可以被用来启动没有配置为自动启动的组件，例如，在初始化时还没启动的组件。
ContextStoppedEvent	当ApplicationContext 停止时发布，在ConfigurableApplicationContext接口上调用stop()方法。”停止”意味这所有的bean的生命周期都会受到一个明确的停止信号。通过调用start()方法可以重启一个已经停止的上下文。
ContextClosedEvent	当ApplicationContext 关闭时发布，在ConfigurableApplicationContext接口上调用close()方法。”关闭”意味着所有的单例bean都会被销毁。关闭的上下文就是它生命周期的末尾。它不能刷新或者重启。
RequestHandledEvent	接受一个HTTP请求的时候，一个特定的web时间会通知所有的bean。这个时间的发布是在请求完成。此事件仅适用于使用Spring的DispatcherServlet的Web应用程序。

你可以创建并发布自己的自定义事件。这个例子演示了一个继承Spring ApplicationEvent的简单类：

```
public class BlackListEvent extends ApplicationEvent {

    private final String address;
    private final String test;

    public BlackListEvent(Object source, String address, String
test) {
        super(source);
        this.address = address;
        this.test = test;
    }

    // accessor and other methods...

}
```

为了发布一个自定义的ApplicationEvent，在ApplicationEventPublisher中调用publishEvent()方法。通常在实现了ApplicationEventPublisherAware接口并把它注册为一个Spring bean的时候它就完成了。下面的例子展示了这么一个类：

```
public class EmailService implements ApplicationEventPublisherAw
are {

    private List<String> blackList;
    private ApplicationEventPublisher publisher;

    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }

    public void setApplicationEventPublisher(ApplicationEventPub
lisher publisher) {
        this.publisher = publisher;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent event = new BlackListEvent(this, addr
ess, text);
            publisher.publishEvent(event);
            return;
        }
        // send email...
    }

}
```

在配置时，Spring容器将检测到EmailService实现了ApplicationEventPublisherAware，并将自动调用setApplicationEventPublisher()方法。实际上，传入的参数将是Spring容器本身;您只需通过ApplicationEventPublisher接口与应用程序上下文进行交互。

为了自定义ApplicationEvent，创建一个试下了ApplicationListener的类并把他注册为一个Spring bean。下面例子展示这样一个类：

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }

}
```

请注意，ApplicationListener通常用你自定义的事件BlackListEvent类型参数化的。这意味着onApplicationEvent()方法可以保持类型安全，避免向下转型的需要。您可以根据需要注册许多的事件侦听器，但请注意，默认情况下，事件侦听器将同步接收事件。这意味着publishEvent()方法会阻塞直到所有的监听者都处理完。这种同步和单线程方法的一个优点是，如果事务上下文可用，它就会在发布者的事务上下文中处理。如果必须需要其他的时间发布策略，请参考javadoc的 Spring ApplicationEventMulticaster 接口。

下面例子展示了使用配置和注册上述每个类的bean定义：

```
<bean id="emailService" class="example.EmailService">
    <property name="blackList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">
    <property name="notificationAddress" value="blacklist@exampl
e.org"/>
</bean>
```

把他们放在一起，当调用emailService的sendEmail()方法时，如果有任何应该被列入黑名单的邮件，那么自定义的BlackListEvent事件会被发布。blackListNotifier 会被注册为一个ApplicationListener，从而接受BlackListEvent，届时通知适当的参与者。

> Spring 的事件机制的设计是用在Spring bean和相同应用上下文的简单通讯。然而，对于更复杂的企业集成需求，单独维护Spring Integration工程对构建著名的Spring编程模型轻量级，面向模式，事件驱动架构提供了完整的支持。

基于注解的事件监听器

从Spring 4.2开始，一个事件监听器可以通过EventListener注解注册在任何managed bean的公共方法上。BlackListNotifier可以重写如下：

```
public class BlackListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddres
s) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlackListEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }

}
```

如上所示，方法签名实际上会推断出它监听的是哪一个类型的事件。这也适用于泛型嵌套，只要你在过滤的时候可以根据泛型参数解析出实际的事件。

如果你的方法需要监听好几个事件或根本没有参数定义它，事件类型也可以用注解本身指明：

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent
.class})
public void handleContextStart() {

}
```

对特殊的时间调用方法，根据定义的SpEL表达式来匹配实际情况，通过条件属性注解，
也可以通过condition注解来添加额外的运行过滤，它对一个特殊事件的方法实际调用是根据它是否匹配condition注解所定义的SpEL表达式。

例如，只要事件的测试属性等于foo，notifier可以被重写为只被调用：

```
@EventListener(condition = "#blEvent.test == 'foo'")
public void processBlackListEvent(BlackListEvent blEvent) {
    // notify appropriate parties via notificationAddress...
}
```

每个SpEL表达式在此评估专用的上下文。下表列出的条目存在上下文中可用，所以可以调用他们处理conditional事件：

表 **3.8.** 存在元数据中的**Event SpEL** 表达式

名字	位置	描述	例子
事件	根路径	实际的ApplicationEvent	#root.event
参数数组	根路径	参数(数组) 目标调用	#root.args[0]
参数名字	上下文	任何的方法参数名称。如果由于某些名称的原因而不可用(例如：没有调试信息)，参数名称也会存在#a<#arg>， #arg代表参数索引开始的地方(从0开始)	#blEvent 或 #a0(也可以使用 #p0 or #p<#arg> 作为别名)

注意，#root.event允许你访问底层的时间，即使你的方法签名实际上是指已发布的任意对象。

如果您需要发布一个事件作为处理另一个事件的结果，只需更改方法签名来返回应该被发布的事件，如下所示：

```
@EventListener
public ListUpdateEvent handleBlackListEvent(BlackListEvent event
) {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}
```

> 异步监听器不支持这个特性

这个新方法将对上述方法处理的每个BlackListEvent都会发布一个新的ListUpdateEvent。如果需要发布多个时间，只需要返回事件集合即可。

异步监听器

如果你希望一个特定的监听器去异步处理事件，只需要重新使用常规的@Async支持：

```
@EventListener
@Async
public void processBlackListEvent(BlackListEvent event) {
    // BlackListEvent is processed in a separate thread
}
```

当使用异步事件的时候有下面两个限制：

1. 如果事件监听器抛出异常，则不会将其传播给调用者，查看AsyncUncaughtExceptionHandler获取详细信息。
2. 此类事件监听器无法发送回复。如果你需要将处理结果发送给另一个时间，注入ApplicationEventPublisher里面手动发送事件。

顺序的监听器

如果你需要一个监听器在另一个监听器调用前被调用，只需要在方法声明上添加@Order注解：

```
@EventListener
@Order(42)
public void processBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress...
}
```

泛型事件

你可以使用泛型来进一步的定义事件的结构。考虑EntityCreatedEvent，T的类型就是你要创建的真实类型。你可以创建下面的监听器定义，它只接受Person类型的EntityCreatedEvent：

```
@EventListener
public void onPersonCreated(EntityCreatedEvent<Person> event) {
...
}
```

触发了事件解析泛型参数，
由于类型擦除，只有在触发了事件解析事件监听过器滤的泛型参数(类似于PersonCreatedEvent继承了EntityCreatedEvent { … })，此操作才会起作用。

在某些情况下，如果所有的时间都遵循相同的结果(上述事件应该是这样)，这可能有点冗余。在这种情况下，你可以实现ResolvableTypeProvider来引导超出框架运行是环境提供的范围：

```
public class EntityCreatedEvent<T>
        extends ApplicationEvent implements ResolvableTypeProvid
er {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(),
                ResolvableType.forInstance(getSource()));
    }
   }
```

> 这不仅适用于ApplicationEvent，还可以作为时间发送的任意对象。

3.15.3 便捷访问低优先级的资源

为了最佳使用和理解上下文，用户应该熟悉Spring资源抽象，如章节：章节4，资源。

一个应用上下文就是一个ResourceLoader，它可以用来载入资源。一个资源本质上讲就是JDK类java.net.URL功能更丰富的版本。实际上，Resource的实现在合适的地方包装了一个java.net.URL实例。资源可以以透明的方式从任何位置获得获取低优先级的资源，包括一个标准的URL，本地文件系统，任何描述标准URL的地方，其他的一些扩展。如果资源位置的字符串是一个没有任何特殊字符前缀的简单路径，那么这些资源就来自特定的并适合实际应用程序上下文类型。

你可以将bean的配置部署到一个实现了应用上下文的特殊回调接口ResourceLoaderAware中，以便在初始化的时候应用上下文把自己作为ResourceLoader传递进去可以自动调用。你也可以暴露资源的属性类型，用于访问静态资源；它们像其他属性一样会被注入。你可以像字符串路径一样指定这些资源的属性，并依赖自动注入上下文的特殊JavaBean的属性编辑器，以便在部署bean时将这些文本字符串转换为真实的对象。

提供给ApplicationContext 构造器的位置路径或路径都是真实的资源字符串，并以简单的形式对特定的上下文进行了适当的处理。ClassPathXmlApplicationContext将简单的路径作为类路径的位置。你也可以使用特殊前缀的位置路径(资源字符串)强制从类路径或者URL加载定义信息，而不管实际的上下文类型。

3.15.4 便捷的ApplicationContext 实例化web 应用程序

你可以通过声明式创建ApplicationContext的实例，例如，ContextLoader。当然你也可以通过使用一个ApplicationContext的实现用编程的方式创建ApplicationContext的实例。

你可以像下面一样通过ContextLoaderListener来注册一个ApplicationContext：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationCon
text.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoade
rListener</listener-class>
</listener>
```

监听器会检查contextConfigLocation的参数。如果参数不存在，监听器默认会使用/WEB-INF/applicationContext.xml。当参数存在的是，监听器会通过预定义的分隔符来（逗号，分号和空格）来分隔字符串，并将其作为应用程序上下文的搜索位置。它也支持Ant风格的路径模式。例如：/WEB-INF/*Context.xml，在WEB-INF目录下，/WEB-INF/**/*Context.xml,WEB-INF子目录的所有这样类似的文件都会被发现。

3.15.5 Spring ApplicationContext 作为Java EE RAR文件部署

可以将Spring ApplicationContext部署为RAR文件，将上下文及其所有必需的bean类和库JAR封装在Java EE RAR部署单元中。这相当于引导一个独立的ApplicationContext，只是托管在Java EE环境中，能够访问Java EE服务器设施。在部署无头WAR文件(实际上，没有任何HTTP入口点，仅用于在Java EE环境中引导Spring ApplicationContext的WAR文件)的情况下RAR部署是更自然的替代方案。RAR部署非常适合不需要HTTP入口点但仅由消息端点和调度作业组成的应用程序上下文。在这种情况下，Bean可以使用应用程序服务器资源，例如JTA事务管理器和JNDI绑定的JDBC DataSources和JMS ConnectionFactory实例，并且还可以通过Spring的标准事务管理和JNDI和JMX支持设施向平台的JMX服务器注册。应用程序组件还可以通过Spring的TaskExecutor抽象实现与应用程序服务器的JCA WorkManager交互。
通过查看 SpringContextResourceAdapter类的JavaDoc，可以知道用于RAR部署中涉及的配置详细信息。

对于Spring ApplicationContext作为Java EE RAR文件的简单部署:将所有应用程序类打包到RAR文件中，这是具有不同文件扩展名的标准JAR文件。将所有必需的库

JAR添加到RAR归档的根目录中。添加一个“META-INF / ra.xml”部署描述符(如SpringContextResourceAdapter的JavaDoc中所示)和相应的Spring XML bean定义文件(通常为“META-INF / applicationContext.xml”)，导致RAR文件进入应用程序服务器的部署目录。

[Note]

这种RAR部署单元通常是独立的; 它们不会将组件暴露给外界，甚至不会暴露给同一应用程序的其他模块。 与基于RAR的ApplicationContext的交互通常通过发生在与其他模块共享的JMS目标的情况下。 基于RAR的ApplicationContext还会在其他情况下使用，例如调度一些作业，对文件系统中的新文件(等等)作出反应。 如果需要允许从外部同步访问，它可以做到如导出RMI端点，然后很自然的可以由同一机器上的其他应用模块使用

可以将Spring ApplicationContext部署为RAR文件，将上下文和所有他所需的bean的类和JAR库封装在Java EE RAR部署单元中。这相当于独立启动一个ApplicationContext，它在Java EE环境中可以访问Java EE服务资源。RAR部署在一些没用头信息的war文件中更自然的选择，实际上，一个war文件在没有http入口的时候，那么它就仅仅是用来在Java EE环境中启动Spring ApplicationContext。

对于不需要HTTP入口点的应用上下文来说RAR部署是一种理想的方式，而不仅是一些消息端点和计划的任务。Bean在这样的上下文中可以使用应用服务器的资源，例如：JTA事务管理器、JNDI-bound JDBC DataSources 和JMS连接工厂实例，也可以通过Spring标准事务管理器、JNDI和JMX支持来注册平台的JMX服务。应用组件也可以通过Spring 抽象TaskExecutor来和应用服务器的JCA WorkManager来进行交互。

有关RAR部署中涉及的配置详细信息请查看 JavaDoc中的SpringContextResourceAdapter。

Spring ApplicationContext 作为Java EE RAR文件的简单部署：将所有的应用类打包进一个RAR文件，它和标准的JAR文件有不同的文件扩展名。将所有需要的库JAR文件添加到RAR归档的根目录中。添加一个”META-INF/ra.xml”部署描述文件（如SpringContextResourceAdapters JavaDoc所示），并更改Spring XML中bean的定义文件（通常为“META-INF / applicationContext.xml”），将生成的rar文件放到音符服务器的部署目录。

> 这种RAR部署单元通常是独立的；它们不会将组建暴露给外界，甚至是同一个应用的其他模块。同基于RAR的ApplicationContext交互通常是通过模块共享的JMS来实现的。一个基于RAR的应用上下文，例如：某些调度任务，文件系统对新文件产生的响应等。如果要允许外界的同步访问，则可以导出RMI端点，这当然可能是同一台机器上的其他应用模块。

3.16 BeanFactory

BeanFactory为Spring的IoC功能提供了底层的基础，但是它仅仅被用于和第三方框架的集成，现在对于大部分的Spring用户来说都是历史了。BeanFactory及其相关的接口，例如：BeanFactoryAware，InitializingBean，DisposableBean，在Spring中仍然有所保留，目的就是为了让大量的第三方框架和Spring集成时保持向后兼容。通常第三方组件不会更加现代的等价物，例如：@PostConstruct 或 @PreDestroy，以便可以与JDK1.4兼容，或避免依赖JSR-250。
这部分提供了BeanFactory 和 ApplicationContext之间的背景差异以及用户怎样通过查找单例的模式来访问IoC容器。

3.16.1 BeanFactory or ApplicationContext？

尽量使用ApplicationContext除非你有更好的理由不用它。
因为ApplicationContext包括了BeanFactory的所有功能，通常也优于BeanFactory,除非一些少数的场景，例如：在受资源约束的嵌入式设备上运行一个嵌入式应用，它的内存消耗可能至关重要，并且可能会产生字节。然而，对于大多数典型的企业级应用和系统来说，ApplicationContext才是你想使用的。Spring大量使用了BeanPostProcessor扩展点（以便使用代理等）。如果你仅仅只使用简单的BeanFactory，很多的支持功能将不会有效，例如：事务和AOP，但至少不会有额外的步骤。这可能会比较迷惑，毕竟配置又没有错。

下表列了BeanFactory 和 ApplicationContext接口和实现的一些特性：

表**3.9** 特性矩阵

Feature	BeanFactory	ApplicationContext
Bean实例化/装配	是	是
BeanPostProcessor自动注册	否	是
BeanFactoryPostProcessor自动注册	否	是
MessageSource便捷访问（针对i18n）	否	是
ApplicationEvent 发布	否	是

用BeanFactory的实现来明确的注册一个bean的后置处理器，你需要写和下面类似的代码：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFact
ory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
factory.addBeanPostProcessor(postProcessor);

// now start using the factory
```

当使用一个BeanFactory的实现来明确的注册一个BeanFactoryPostProcessor时，你写的代码必须和下面类似：

```
DefaultListableBeanFactory factory = new DefaultListableBeanFact
ory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(fac
tory);
reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfi
gurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

在这两种情况下，明确的注册步不是很方便，这也就是为什么在大多数支持Spring的应用中，ApplicationContext的各种实现都优于BeanFactory实现的原因之一，特别是当使用BeanFactoryPostProcessors和BeanPostProcessors的时候。这些机制实现了一些很重要的功能，例如：属性的占位替换和AOP。

4. 资源

4.1 介绍

仅仅使用 JAVA 的 java.net.URL 和针对不同 URL 前缀的标准处理器，并不能满足我们对各种底层资源的访问，比如：我们就不能通过 URL 的标准实现来访问相对类路径或者相对 ServletContext 的各种资源。虽然我们可以针对特定的 url 前缀来注册一个新的 URLStreamHandler（和现有的针对各种特定前缀的处理器类似，比如 http：），然而这往往会是一件比较麻烦的事情(要求了解 url 的实现机制等），而且 url 接口也缺少了部分基本的方法，如检查当前资源是否存在的方法。

4.2 Resource 接口

相对标准 url 访问机制，spring 的 Resource 接口对抽象底层资源的访问提供了一套更好的机制。

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOExcept
ion;

    String getFilename();

    String getDescription();

}
```

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;

}
```

Resource 接口里的最重要的几个方法：

- getInputStream(): 定位并且打开当前资源，返回当前资源的 InputStream。预计每一次调用都会返回一个新的 – InputStream，因此关闭当前输出流就成为了调用者的责任。
- exists(): 返回一个 boolean，表示当前资源是否真的存在。

- isOpen(): 返回一个 boolean，表示当前资源是否一个已打开的输入流。如果结果为 true，返回的 InputStream 不能多次读取，只能是一次性读取之后，就关闭 InputStream，以防止内存泄漏。除了 InputStreamResource，其他常用 Resource 实现都会返回 false。
- getDescription(): 返回当前资源的描述，当处理资源出错时，资源的描述会用于错误信息的输出。一般来说，资源的描述是一个完全限定的文件名称，或者是当前资源的真实 url。

Resource 接口里的其他方法可以让你获得代表当前资源的 URL 或 File 对象（前提是底层实现可兼容的，也支持该功能）。

Resource抽象在Spring本身被广泛使用，作为需要资源的许多方法签名中的参数类型。 某些Spring API中的其他方法（例如各种ApplicationContext实现的构造函数）采用一个String，它以未安装或简单的形式用于创建适用于该上下文实现的资源，或者通过String路径上的特殊前缀，允许调用者 以指定必须创建和使用特定的资源实现。

Resource 接口（实现）不仅可以被 spring 大量的应用，其也非常适合作为你编程中访问资源的辅助工具类。当你仅需要使用到 Resource 接口实现时，可以直接忽略 spring 的其余部分。单独使用 Rsourece 实现，会造成代码与 spring 的部分耦合，可也仅耦合了其中一小部分辅助类，而且你可以将 Reource 实现作为 URL 的一种访问底层更为有效的替代，与你引入其他库来达到这种目的是一样的。

需要注意的是 Resource 实现并没有去重新发明轮子，而是尽可能地采用封装。举个例子，UrlResource 里就封装了一个 URL 对象，在其内的逻辑就是通过封装的 URL 对象来完成的。

4.3 内置的 Resource 实现

spring 直接提供了多种开箱即用的 Resource 实现。

4.3.1 UrlResource

UrlResource 封装了一个 java.net.URL 对象，用来访问 URL 可以正常访问的任意对象，比如文件、an HTTP target, an FTP target, 等等。所有的 URL 都可以用一个标准化的字符串来表示。如通过正确的标准化前缀，可以用来表示当前 URL 的类型，当中就包括用于访问文件系统路径的 file:,通过 http 协议访问资源的 http:,通过 ftp 协议访问资源的 ftp:，还有很多......

可以显式化地使用 UrlResource 构造函数来创建一个 UrlResource，不过通常我们可以在调用一个 api 方法是，使用一个代表路径的 String 参数来隐式创建一个 UrlResource。对于后一种情况，会由一个 javabean PropertyEditor 来决定创建哪一种 Resource。如果路径里包含某一个通用的前缀（如 classpath:),PropertyEditor 会根据这个通用的前缀来创建恰当的 Resource；反之，如果 PropertyEditor 无法识别这个前缀，会把这个路径作为一个标准的 URL 来创建一个 UrlResource。

4.3.2 ClassPathResource

ClassPathResource 可以从类路径上加载资源，其可以使用线程上下文加载器、指定加载器或指定的 class 类型中的任意一个来加载资源。

当类路径上资源存于文件系统中，ClassPathResource 支持以 java.io.File 的形式访问，可当类路径上的资源存于尚未解压(没有 被Servlet 引擎或其他可解压的环境解压）的 jar 包中，ClassPathResource 就不再支持以 java.io.File 的形式访问。鉴于上面所说这个问题，spring 中各式 Resource 实现都支持以 jave.net.URL 的形式访问。

可以显式使用 ClassPathResource 构造函数来创建一个 ClassPathResource ，不过通常我们可以在调用一个 api 方法时，使用一个代表路径的 String 参数来隐式创建一个 ClassPathResource。对于后一种情况，会由一个 javabean PropertyEditor 来识别路径中 classpath: 前缀，从而创建一个 ClassPathResource。

4.3.3 FileSystemResource

这是针对 java.io.File 提供的 Resource 实现。显然，我们可以使用 FileSystemResource 的 getFile() 函数获取 File 对象，使用 getURL() 获取 URL 对象。

4.3.4 ServletContextResource

这是为了获取 web 根路径的 ServletContext 资源而提供的 Resource 实现。

ServletContextResource 完全支持以流和 URL 的方式访问，可只有当 web 项目是已解压的(不是以 war 等压缩包形式存在)且该 ServletContext 资源存于文件系统里，ServletContextResource 才支持以 java.io.File 的方式访问。至于说到，我们的 web 项目是否已解压和相关的 ServletContext 资源是否会存于文件系统里，这个取决于我们所使用的 Servlet 容器。若 Servlet 容器没有解压 web 项目，我们可以直接以 JAR 的形式的访问，或者其他可以想到的方式（如访问数据库）等。

4.3.5 InputStreamResource

这是针对 InputStream 提供的 Resource 实现。建议，在确实没有找到其他合适的 Resource 实现时，才使用 InputSteamResource。如果可以，尽量选择 ByteArrayResource 或其他基于文件的 Resource 实现来代替。

与其他 Resource 实现已比较，InputStreamRsource 倒像一个已打开资源的描述符,因此，调用 isOpen() 方法会返回 true。除了在需要获取资源的描述符或需要从输入流多次读取时，都不要使用 InputStreamResource 来读取资源。

4.3.6 ByteArrayResource

这是针对字节数组提供的 Resource 实现。可以通过一个字节数组来创建 ByteArrayResource。

当需要从字节数组加载内容时，ByteArrayResource 是一个不错的选择，使用 ByteArrayResource 可以不用求助于 InputStreamResource。

4.4 ResourceLoader 接口

ResourceLoader 接口是用来加载 Resource 对象的，换句话说，就是当一个对象需要获取 Resource 实例时，可以选择实现 ResourceLoader 接口。

```
public interface ResourceLoader {

    Resource getResource(String location);

}
```

spring 里所有的应用上下文都是实现了 ResourceLoader 接口，因此，所有应用上下文都可以通过 getResource() 方法获取 Resource 实例。

当你在指定应用上下文调用 getResource() 方法时，而指定的位置路径又没有包含特定的前缀，spring 会根据当前应用上下文来决定返回哪一种类型 Resource。举个例子，假设下面的代码片段是通过 ClassPathXmlApplicationContext 实例来调用的，

```
Resource template = ctx.getResource("some/resource/path/myTempla
te.txt");
```

那 spring 会返回一个 ClassPathResource 对象；类似的，如果是通过实例 FileSystemXmlApplicationContext 实例调用的，返回的是一个 FileSystemResource 对象；如果是通过 WebApplicationContext 实例的，返回的是一个 ServletContextResource 对象...... 如上所说，你就可以在指定的应用上下中使用 Resource 实例来加载当前应用上下文的资源。

还有另外一种场景里，如在其他应用上下文里，你可能会强制需要获取一个 ClassPathResource 对象，这个时候，你可以通过加上指定的前缀来实现这一需求，如：

```
Resource template = ctx.getResource("classpath:some/resource/pat
h/myTemplate.txt");
```

类似的，你可以通过其他任意的 url 前缀来强制获取 UrlResource 对象：

```
Resource template = ctx.getResource("file:///some/resource/path/
myTemplate.txt");
```

```
Resource template = ctx.getResource("http://myhost.com/resource/
path/myTemplate.txt");
```

下面，给出一个表格来总结一下 spring 根据各种位置路径加载资源的策略：

Table 4.1. Resource strings

前缀	例子	解释
classpath:	classpath:com/myapp/config.xml	从类路径加载
file:	file:///data/config.xml	以URL形式从文件系统加载
http:	http://myserver/logo.png	以URL形式加载
(none)	/data/config.xml	由底层的ApplicationContext实现决定

4.5 ResourceLoaderAware 接口

ResourceLoaderAware 是一个特殊的标记接口，用来标记提供 ResourceLoader 引用的对象。

```
public interface ResourceLoaderAware {

    void setResourceLoader(ResourceLoader resourceLoader);
}
```

当将一个 ResourceLoaderAware 接口的实现类部署到应用上下文时(此类会作为一个 spring 管理的 bean）, 应用上下文会识别出此为一个 ResourceLoaderAware 对象，并将自身作为一个参数来调用 setResourceLoader() 函数，如此，该实现类便可使用 ResourceLoader 获取 Resource 实例来加载你所需要的资源。（附：为什么能将应用上下文作为一个参数来调用 setResourceLoader() 函数呢？不要忘了，在前文有谈过，spring 的所有上下文都实现了 ResourceLoader 接口）。

当然了，一个 bean 若想加载指定路径下的资源，除了刚才提到的实现 ResourcesLoaderAware 接口之外（将 ApplicationContext 作为一个 ResourceLoader 对象注入），bean 也可以实现 ApplicationContextAware 接口，这样可以直接使用应用上下文来加载资源。但总的来说，在需求满足都满足的情况下，最好是使用的专用 ResourceLoader 接口，因为这样代码只会与接口耦合，而不会与整个 spring ApplicationContext 耦合。与 ResourceLoader 接口耦合，抛开 spring 来看，就是提供了一个加载资源的工具类接口。

从 spring 2.5 开始，除了实现 ResourceLoaderAware 接口，也可采取另外一种替代方案——依赖于 ResourceLoader 的自动装配。”传统”的 constructor 和 bytype 自动装配模式都支持 ResourceLoader 的装配（可参阅 Section 5.4.5, “自动装配协作者”）——前者以构造参数的形式装配，后者以 setter 方法中参数装配。若为了获得更大的灵活性(包括属性注入的能力和多参方法)，可以考虑使用基于注解的新注入方式。使用注解 @Autowiring 标记 ResourceLoader 变量，便可将其注入到成员属性、构造参数或方法参数中(@autowiring 详细的使用方法可参考Section 3.9.2, “@Autowired”.)。

4.6 资源依赖

如果bean本身将通过某种动态过程来确定和提供资源路径，那么bean可以使用ResourceLoader接口来加载资源。 j假设以某种方式加载一个模板，其中需要的特定资源取决于用户的角色。 如果资源是静态的，那么完全消除ResourceLoader接口的使用是有意义的，只需让bean公开它需要的Resource属性，那么它们就会以你所期望的方式被注入。

什么使得它们轻松注入这些属性，是所有应用程序上下文注册和使用一个特殊的JavaBeans PropertyEditor，它可以将String路径转换为Resource对象。 因此，如果myBean具有“资源”类型的模板属性，则可以使用该资源的简单字符串进行配置，如下所示：

```
<bean id="myBean" class="...">
<property name="template" value="some/resource/path/myTemplate.t
xt"/>
</bean>
```

请注意，资源路径没有前缀，因为应用程序上下文本身将用作ResourceLoader，资源本身将通过ClassPathResource，FileSystemResource或ServletContextResource（根据需要）加载，具体取决于上下文的确切类型。

如果需要强制使用特定的资源类型，则可以使用前缀。 以下两个示例显示如何强制使用ClassPathResource和UrlResource（后者用于访问文件系统文件）。

```
<property name="template" value="classpath:some/resource/path/my
Template.txt">
<property name="template" value="file:///some/resource/path/myTe
mplate.txt"/>
```

4.7 应用上下文和资源路径

4.7.1 构造应用上下文

（某一特定）应用上下文的构造器通常可以使用字符串或字符串数组所指代的(多个)资源(如 xml 文件)来构造当前上下文。

当指定的位置路径没有带前缀时，那从指定位置路径创建的 Resource 类型(用于后续加载 bean 定义),取决于所使用应用上下文。举个列子，如下所创建的 ClassPathXmlApplicationContext ：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

会从类路径加载 bean 的定义，因为所创建的 Resource 实例是 ClassPathResource.但所创建的是 FileSystemXmlApplicationContext 时，

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("conf/appContext.xml");
```

则会从文件系统加载 bean 的定义，这种情况下，资源路径是相对工作目录而言的。

注意：若位置路径带有 classpath 前缀或 URL 前缀，会覆盖默认创建的用于加载 bean 定义的 Resource 类型，比如这种情况下的 FileSystemXmlApplicationContext

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

，实际是从类路径下加载了 bean 的定义。可是，这个上下文仍然是 FileSystemXmlApplicationContext，而不是 ClassPathXmlApplicationContext，在后续作为 ResourceLoader 来使用时，不带前缀的路径仍然会从文件系统中加载。

构造 **ClassPathXmlApplicationContext** 实例 **–** 快捷方式

ClassPathXmlApplicationContext 提供了多个构造函数，以利于快捷创建 ClassPathXmlApplicationContext 的实例。最好莫不过使用只包含多个 xml 文件名（不带路径信息）的字符串数组和一个 Class 参数的构造器，所省略路径信息 ClassPathXmlApplicationContext 会从 Class 参数 获取：

下面的这个例子，可以让你对个构造器有比较清晰的认识。试想一个如下类似的目录结构：

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

由 services.xml 和 daos.xml 中 bean 所组成的 ClassPathXmlApplicationContext，可以这样来初始化：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
                new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

欲要知道 ClassPathXmlApplicationContext 更多不同类型的构造器，请查阅 Javadocs 文档。

4.7.2 使用通配符构造应用上下文

从前文可知，应用上下文构造器的中的资源路径可以是单一的路径（即一对一地映射到目标资源）；另外资源路径也可以使用高效的通配符——可包含 classpath*：前缀 或 ant 风格的正则表达式（使用 spring 的 PathMatcher 来匹配）。

通配符机制的其中一种应用可以用来组装组件式的应用程序。应用程序里所有组件都可以在一个共知的位置路径发布自定义的上下文片段，则最终应用上下文可使用 classpath*: 在同一路径前缀(前面的共知路径）下创建，这时所有组件上下文的片段都会被自动组装。

谨记，路径中的通配符特定用于应用上下文的构造器，只会在应用构造时有效，与其 Resource 自身类型没有任何关系。不可以使用 classpth*：来构造任一眞实的 Resource，因为一个资源点一次只可以指向一个资源。（如果直接使用 PathMatcher 的工具类，也可以在路径中使用通配符）

Ant 风格模式

以下是一些使用了 Ant 风格的位置路径：

```
/WEB-INF/*-context.xml
  com/mycompany/**/applicationContext.xml
  file:C:/some/path/*-context.xml
  classpath:com/mycompany/**/applicationContext.xml
```

当位置路径使用了 ant 风格，解释器会遵循一套复杂且预定义的逻辑来解释这些位置路径。解释器会先从位置路径里获取最靠前的不带通配符的路径片段，使用这个路径片段来创建一个 Resource ，并从 Resource 里获取其 URL，若所获取到 URL 前缀并不是 “jar:”,或其他特殊容器产生的特殊前缀（如 WebLogic 的 zip:,WebSphere wsjar),则从 Resource 里获取 java.io.File 对象，并通过其遍历文件系统。进而解决位置路径里通配符;若获取的是 “jar:”的 URL ，解析器会从其获取一个 java.net.JarURLConnection 或手动解析此 URL，并遍历 jar 文件的内容进而解决位置路径的通配符。

对可移植性的影响

如果指定的路径已经是文件URL（显式地或隐含地，因为基本的ResourceLoader是一个文件系统的，那么通配符将保证以完全可移植的方式工作。

如果指定的路径是类路径位置，则解析器必须通过Classloader.getResource（）调用获取最后一个非通配符路径段URL。 由于这只是路径的一个节点（而不是最后的文件），在这种情况下，它实际上是未定义的（在ClassLoader javadocs中）返回的是什么样的URL。 实际上，它始终是一个java.io.File，它表示类路径资源解析为文件系统位置的目录或某种类型的jar URL，其中类路径资源解析为一个jar位置。尽管如此，这种操作仍然存在可移植性问题。

如果为最后一个非通配符段获取了一个jar URL，解析器必须能够从中获取 java.net.JarURLConnection，或者手动解析jar URL，以便能够遍历该jar的内容，然后解析 通配符。 这将在大多数环境中正常工作，但在其他环境中将会失败，并

且强烈建议您在依赖它之前，彻底地在您的特定环境中彻底测试来自jar的资源的通配符解析。

classpath*: 的可移植性

当构造基于 xml 文件的应用上下文时，位置路径可以使用 classpath*：前缀：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("cla
sspath*:conf/appContext.xml");
```

classpath*：的使用表示类路径下所有匹配文件名称的资源都会被获取(本质上就是调用了 ClassLoader.getResources(…) 方法），接着将获取到的资源组装成最终的应用上下文。

> 通配符路径依赖了底层 classloader 的 getResource 方法。可是现在大多数应用服务器提供了自身的 classloader 实现，其处理 jar 文件的形式可能各有不同。要在指定服务器测试 classpath*: 是否有效，简单点可以使用 getClass().getClassLoader().getResources(“”) 去加载类路径 jar包里的一个文件。尝试在两个不同的路径加载名称相同的文件，如果返回的结果不一致，就需要查看一下此服务器中与 classloader 行为设置相关的文档。

在位置路径的其余部分，classpath*: 前缀可以与 PathMatcher 结合使用，如：” classpath*:META-INF/*-beans.xml”。这种情况的解析策略非常简单：取位置路径最靠前的无通配符片段，调用 ClassLoader.getResources() 获取所有匹配的类层次加载器可加载的的资源，随后将 PathMacher 的策略应用于每一个获得的资源（起过滤作用）。

通配符的补充说明

除非所有目标资源都存于文件系统，否则classpath*：和 ant 风格模式的结合使用，都只能在至少有一个确定根包路径的情况下，才能达到预期的效果。换句话说，就是像 classpath*:*.xml 这样的 pattern 不能从根目录的 jar 文件中获取资源，只能从根目录的扩展目录获取资源。此问题的造成源于 jdk ClassLoader.getResources() 方法的局限性——当向 ClassLoader.getResources() 传入空串时(表示搜索潜在的根目录)，只能获取的文件系统的文件位置路径，即获取不了 jar 中文件的位置路径。

如果在多个类路径上存在所搜索的根包，那使用 classpath: 和 ant 风格模式一起指定的资源不保证找到匹配的资源。因为使用如下的 pattern
classpath:com/mycompany/**/service-context.xml
去搜索只在某一个路径存在的指定资源com/mycompany/package1/service-context.xml
时,解析器只会对 getResource(“com/mycompany”) 返回的(第一个) URL 进行遍历和解释，则当在多个类路径存在基础包节点 “com/mycompany” 时(如在多个 jar 存在这个基础节点),解析器就不一定会找到指定资源。因此，这种情况下建议结合使用 classpath*: 和 ant 风格模式，classpath*：会让解析器去搜索所有包含基础包节点的类路径。

4.7.3 FileSystemResource 注意事项

FileSystemResource 没有依附 FileSystemApplicationContext，因为 FileSystemApplicationContext 并不是一个真正的 `ResourceLoader。FileSystemResource 并没有按约定规则来处理绝对和相对路径。相对路径是相对与当前工作而言，而绝对路径则是相对文件系统的根目录而言。

然而为了向后兼容，当 FileSystemApplicationContext 是一个 ResourceLoader 实例时，我们做了一些改变 —— 不管 FileSystemResource` 实例的位置路径是否以 / 开头， FileSystemApplicationContext 都强制将其作为相对路径来处理。事实上，这意味着以下例子等效：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("/conf/context.xml");
```

还有：（即使它们的意义不一样 —— 一个是相对路径，另一个是绝对路径。）

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

实践中，如果确实需要使用绝对路径，建议放弃 FileSystemResource / FileSystemXmlApplicationContext 在绝对路劲的使用，而强制使用 file: 的 UrlResource。

```
// Resource 只会是 UrlResource，与上下文的真实类型无关
ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

```
// 强制 FileSystemXmlApplicationContext 通过 UrlResource 加载资源
ApplicationContext ctx = new FileSystemXmlApplicationContext("fi
le:///conf/context.xml");
```

5. 验证、数据绑定和类型转换

5.1 介绍

JSR-303/JSR-349 Bean Validation

在设置支持方面，Spring Framework 4.0支持Bean Validation 1.0(JSR-303)和Bean Validation 1.1(JSR-349)，也将其改写成了Spring的 `Validator` 接口。

正如5.8 Spring验证所述，应用程序可以选择一次性全局启用Bean验证，并使其专门用于所有的验证需求。

正如5.8.3 配置DataBinder所述，应用程序也可以为每个 `DataBinder` 实例注册额外的Spring `Validator` 实例，这可能有助于不通过使用注解而插入验证逻辑。

考虑将验证作为业务逻辑是有利有弊的，Spring提供了一种不排除利弊的用于验证(和数据绑定)的设计。具体的验证不应该捆绑在web层，应该容易本地化并且它应该能够插入任何可用的验证器。考虑到以上这些，Spring想出了一个 `Validator` 接口，它在应用程序的每一层基本都是可用的。数据绑定对于将用户输入动态绑定到应用程序的领域模型上(或者任何你用于处理用户输入的对象)是非常有用的。Spring提供了所谓的 `DataBinder` 来处理这个。 `Validator` 和 `DataBinder` 组成了 `validation` 包，其主要用于但并不局限于MVC框架。

`BeanWrapper` 是Spring框架中的一个基本概念且在很多地方使用。然而，你可能并不需要直接使用 `BeanWrapper` 。尽管这是参考文档，我们仍然觉得有一些说明需要一步步来。我们将会在本章中解释 `BeanWrapper` ，因为你极有可能会在尝试将数据绑定到对象的时候使用它。

Spring的DataBinder和底层的BeanWrapper都使用PropertyEditor来解析和格式化属性值。 `PropertyEditor` 概念是JavaBeans规范的一部分，并会在本章进行说明。Spring 3不仅引入了"core.convert"包来提供一套通用类型转换工具，还有一个高层次的"format"包用于格式化UI字段值。可以将这些新包视作更简单的PropertyEditor替代方式来使用，本章还会对此进行讨论。

5.2 使用Spring的验证器接口进行验证

Spring具有一个 `Validator` 接口可以让你用于验证对象。 `Validator` 接口在工作时需要使用一个 `Errors` 对象，以便于在验证过程中，验证器可以将验证失败的信息报告给这个 `Errors` 对象。

让我们考虑一个小的数据对象：

```
public class Person {

    private String name;
    private int age;

    // the usual getters and setters...
}
```

通过实现 `org.springframework.validation.Validator` 的下列两个接口，我们打算为 `Person` 类提供验证行为：

- `support(Class)` — 这个 `Validator` 是否可以验证给定 `Class` 的实例 `validate(Object,org.springframework.validation.Errors)`
- — 验证给定的对象并且万一验证错误，可以将这些错误注册到给定的 `Errors` 对象

实现一个 `Validator` 是相当简单的，特别是当你知道Spring框架还提供了 `ValidationUtils` 辅助类：

```
public class PersonValidator implements Validator {

    /**
     * This Validator validates *just* Person instances
     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

正如你看到的，`ValidationUtils` 类的静态方法 `rejectIfEmpty(..)` 被用于拒绝那些值为 `null` 或者空字符串的 `'name'` 属性。除了上面展示的例子之外，去看一看 `ValidationUtils` 的java文档有助于了解它提供的功能。

通过实现单个的 `Validator` 类来逐个验证富对象中的嵌套对象当然是有可能的，然而将验证逻辑封装在每个嵌套类对象自身的 `Validator` 实现中可能是一种更好的选择。`Customer` 就是一个'富'对象的简单示例，它由两个字符串属性(姓和名)以及一个复杂对象 `Address` 组成。`Address` 对象可能独立于 `Customer` 对象使用，因此已经实现了一个独特的 `AddressValidator` 。如果你想要你的 `CustomerValidator` 不借助于复制粘贴而重用包含在 `AddressValidator` 中的逻辑，那么你可以通过依赖注入或者实例化你的 `CustomerValidator` 中的 `AddressValidator` ，然后像这样使用它：

```
public class CustomerValidator implements Validator {

    private final Validator addressValidator;

    public CustomerValidator(Validator addressValidator) {
```

```
        if (addressValidator == null) {
            throw new IllegalArgumentException("The supplied [Va
lidator] is " +
                "required and must not be null.");
        }
        if (!addressValidator.supports(Address.class)) {
            throw new IllegalArgumentException("The supplied [Va
lidator] must " +
                "support the validation of [Address] instances."
);
        }
        this.addressValidator = addressValidator;
    }

    /**
     * This Validator validates Customer instances, and any subc
lasses of Customer too
     */
    public boolean supports(Class clazz) {
        return Customer.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "first
Name", "field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surna
me", "field.required");
        Customer customer = (Customer) target;
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidato
r, customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
}
```

验证错误被报告给传递到验证器的 `Errors` 对象。在使用Spring Web MVC的情况下，你可以使用 `<spring:bind/>` 标签来检查错误信息，不过当然你也可以自己检查错误对象。有关它提供的方法的更多信息可以在java文档中找到。

5.3 将代码解析成错误消息

在之前我们已经谈论了数据绑定和验证，最后一件值得讨论的事情是输出对应于验证错误的消息。在我们上面展示的例子里，我们拒绝了 `name` 和 `age` 字段。如果我们要使用 `MessageSource` 来输出错误消息，我们将会使用我们在拒绝该字段(这个情况下是'姓名'和'年龄')时给出的错误代码。当你调用(不管是直接调用还是间接通过使用 `ValidationUtils` 类调用)来自 `Errors` 接口的 `rejectValue` 或者其他 `reject` 方法时，其底层实现不仅会注册你传入的代码，还会注册一些额外的错误代码。注册怎样的错误代码取决于它所使用的 `MessageCodesResolver` ，默认情况下，会使用 `DefaultMessageCodesResolver` ，其不仅会使用你提供的代码注册消息，还会注册包含你传递给拒绝方法的字段名称的消息。所以如果你使用 `rejectValue("age", "too.darn.old")` 来拒绝一个字段，除了 `too.darn.old` 代码，Spring还会注册 `too.darn.old.age` 和 `too.darn.old.age.int` (第一个会包含字段名称且第二个会包含字段类型)。这样做是为了方便开发人员来定位错误消息等。

有关 `MessageCodesResolver` 和其默认策略的更多信息可以分别在 `MessageCodesResolver` 以及 `DefaultMessageCodesResolver` 的在线java文档中找到。

5.4 Bean操作和BeanWrapper

`org.springframework.beans` 包遵循Oracle提供的JavaBeans标准。一个JavaBean只是一个包含默认无参构造器的类，它遵循一个命名约定(通过一个例子)：一个名为 `bingoMadness` 属性将有一个设置方法 `setBingoMadness(..)` 和一个获取方法 `getBingoMadness(..)` 。有关JavaBeans和其规范的更多信息，请参考Oracle的网站(javabeans)。

beans包里一个非常重要的类是 `BeanWrapper` 接口和它的相应实现(`BeanWrapperImpl`)。引用自java文档， `BeanWrapper` 提供了设置和获取属性值(单独或批量)、获取属性描述符以及查询属性以确定它们是可读还是可写的功能。 `BeanWrapper` 还提供对嵌套属性的支持，能够不受嵌套深度的限制启用子属性的属性设置。然后， `BeanWrapper` 提供了无需目标类代码的支持就能够添加标准JavaBeans的 `PropertyChangeListeners` 和 `VetoableChangeListeners` 的能力。最后然而并非最不重要的是， `BeanWrapper` 提供了对索引属性设置的支持。 `BeanWrapper` 通常不会被应用程序的代码直接使用，而是由 `DataBinder` 和 `BeanFactory` 使用。

`BeanWrapper` 的名字已经部分暗示了它的工作方式：它包装一个bean以对其执行操作，比如设置和获取属性。

5.4.1 设置并获取基本和嵌套属性

使用 `setPropertyValue(s)` 和 `getPropertyValue(s)` 可以设置并获取属性，两者都带有几个重载方法。在Spring自带的java文档中对它们有更详细的描述。重要的是要知道对象属性指示的几个约定。几个例子：

表 **5.1.** 属性示例

表达式	说明
`name`	表示属性 `name` 与方法 `getName()` 或 `isName()` 和 `setName()` 相对应
`account.name`	表示属性 account 的嵌套属性 name 与方法 getAccount().setName() 或 getAccount().getName() 相对应
`account[2]`	表示索引属性 `account` 的第三个元素。索引属性可以是 `array` 、 `list` 或其他自然排序的集合
`account[COMPANYNAME]`	表示映射属性 account 被键*COMPANYNAME*索引的值

下面你会发现一些使用 `BeanWrapper` 来获取和设置属性的例子。

*(*如果你不打算直接使用 `BeanWrapper` ，那么下一部分对你来说并不重要。如果你仅使用 `DataBinder` 和 `BeanFactory` 以及它们开箱即用的实现，你应该跳到关于 `PropertyEditor` 部分的开头*)*。

考虑下面两个类：

```
public class Company {

    private String name;
    private Employee managingDirector;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Employee getManagingDirector() {
        return this.managingDirector;
    }

    public void setManagingDirector(Employee managingDirector) {
        this.managingDirector = managingDirector;
    }
}
```

```
public class Employee {

    private String name;

    private float salary;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

以下的代码片段展示了如何检索和操纵实例化的 `Companies` 和 `Employees` 的某些属性：

```
BeanWrapper company = new BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company In
c.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = new BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInsta
nce());

// retrieving the salary of the managingDirector through the com
pany
Float salary = (Float) company.getPropertyValue("managingDirecto
r.salary");
```

5.4.2 内置PropertyEditor实现

Spring使用`PropertyEditor`的概念来实现`Object`和`String`之间的转换。如果你考虑到它，有时候换另一种方式表示属性可能比对象本身更方便。举个例子，一个`Date`可以以人类可读的方式表示(如`String'2007-14-09'`)，同时我们依然能把人类可读的形式转换回原始的时间(甚至可能更好：将任何以人类可读形式输入的时间转换回`Date`对象)。这种行为可以通过注册类型为`PropertyEditor`的自定义编辑器来实现。在`BeanWrapper`或上一章提到的特定IoC容器中注册自定义编辑器，可以使其了解如何将属性转换为期望的类型。请阅读Oracle为`java.beans`包提供的java文档来获取更多关于`PropertyEditor`的信息。

这是Spring使用属性编辑的两个例子：

- 使用`PropertyEditor`来完成*bean*的属性设置。当提到将`java.lang.String`作为你在XML文件中声明的某些bean的属性值时，Spring将会(如果相应的属性的设置方法具有一个`Class`参数)使用`ClassEditor`尝试将参数解析成`Class`对象。
- 在Spring的MVC框架中解析*HTTP*请求的参数是由各种`PropertyEditor`完成的，你可以把它们手动绑定到`CommandController`的所有子类。

Spring有一些内置的 `PropertyEditor` 使生活变得轻松。它们中的每一个都已列在下面，并且它们都被放在 `org.springframework.beans.propertyeditors` 包中。大部分但并不是全部(如下所示)，默认情况下会由 `BeanWrapperImpl` 注册。在某种方式下属性编辑器是可配置的，那么理所当然，你可以注册你自己的变种来覆盖默认编辑器：

Table 5.2. 内置PropertyEditor

类	说明
`ByteArrayPropertyEditor`	针对字节数组的编辑器。字符串会简单地转换成相应的字节表示。默认情况下由 `BeanWrapperImpl` 注册。
`ClassEditor`	将类的字符串表示形式解析成实际的类形式并且也能返回实际类的字符串表示形式。如果找不到类，会抛出一个 `IllegalArgumentException` 。默认情况下由 `BeanWrapperImpl` 注册。
`CustomBooleanEditor`	针对 `Boolean` 属性的可定制的属性编辑器。默认情况下由 `BeanWrapperImpl` 注册，但是可以作为一种自定义编辑器通过注册其自定义实例来进行覆盖。
`CustomCollectionEditor`	针对集合的属性编辑器，可以将原始的 `Collection` 转换成给定的目标 `Collection` 类型。
`CustomDateEditor`	针对java.util.Date的可定制的属性编辑器，支持自定义的时间格式。不会被默认注册，用户必须使用适当格式进行注册。
`CustomNumberEditor`	针对任何Number子类(比如 `Integer` 、 `Long` 、 `Float` 、 `Double`)的可定制的属性编辑器。默认情况下由 `BeanWrapperImpl` 注册，但是可以作为一种自定义编辑器通过注册其自定义实例来进行覆盖。
`FileEditor`	能够将字符串解析成 `java.io.File` 对象。默认情况下由 `BeanWrapperImpl` 注册。
`InputStreamEditor`	一次性的属性编辑器，能够读取文本字符串并生成(通过中间的 `ResourceEditor` 以及 `Resource`)一个 `InputStream` 对象，因此 `InputStream` 类型的属性可以直接以字符串设置。请注意默认的使用方式不会为你关

	闭 `InputStream` ！默认情况下由 `BeanWrapperImpl` 注册。
`LocaleEditor`	能够将字符串解析成 `Locale` 对象，反之亦然(字符串格式是*[country]*[variant]，这与Locale提供的toString()方法是一样的)。默认情况下由 `BeanWrapperImpl` 注册。
`PatternEditor`	能够将字符串解析成 `java.util.regex.Pattern` 对象，反之亦然。
`PropertiesEditor`	能够将字符串(按照 `java.util.Properties` 类的java文档定义的格式进行格式化)解析成 `Properties` 对象。默认情况下由 `BeanWrapperImpl` 注册。
`StringTrimmerEditor`	用于缩减字符串的属性编辑器。有选择性允许将一个空字符串转变成 `null` 值。不会进行默认注册，需要在用户有需要的时候注册。
`URLEditor`	能够将一个URL的字符串表示解析成实际的 `URL` 对象。默认情况下由 `BeanWrapperImpl` 注册。

Spring使用 `java.beans.PropertyEditorManager` 来设置可能需要的属性编辑器的搜索路径。搜索路径中还包括了 `sun.bean.editors`

，这个包里面包含如 `Font` 、 `Color` 类型以及其他大部分基本类型的 `PropertyEditor` 实现。还要注意的是，如果 `PropertyEditor` 类与它们所处理的类位于同一个包并且除了'Editor'后缀之外拥有相同的名字，那么标准的JavaBeans基础设施会自动发现这些它们(不需要你显式的注册它们)。例如，有人可能会有以下的类和包结构，这已经足够识别出 `FooEditor` 类并将其作为 `Foo` 类型属性的 `PropertyEditor` 。

```
com
  chank
    pop
      Foo
      FooEditor // the PropertyEditor for the Foo class
```

要注意的是在这里你也可以使用标准JavaBeans机制的 `BeanInfo` (在in not-amazing-detail here有描述)。在下面的示例中，你可以看使用 `BeanInfo` 机制为一个关联类的属性显式注册一个或多个 `PropertyEditor` 实例。

```
com
  chank
    pop
      Foo
      FooBeanInfo // the BeanInfo for the Foo class
```

这是被引用到的 `FooBeanInfo` 类的Java源代码。它会将一个 `CustomNumberEditor` 同 `Foo` 类的 `age` 属性关联。

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEdit
or(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescr
iptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Objec
t bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

注册额外的自定义PropertyEditor

当bean属性设置成一个字符串值时，Spring IoC容器最终会使用标准JavaBeans的 `PropertyEditor` 将这些字符串转换成复杂类型的属性。Spring预先注册了一些自定义 `PropertyEditor` (例如将一个以字符串表示的类名转换成真正

的 `Class` 对象)。此外，Java的标准JavaBeans `PropertyEditor` 查找机制允许一个 `PropertyEditor` 只需要恰当的命名并同它支持的类位于相同的包，就能够自动发现它。

如果需要注册其他自定义的 `PropertyEditor` ，还有几种可用机制。假设你有一个 `BeanFactory` 引用，最人工化的方式(但通常并不方便或者推荐)是直接使用 `ConfigurableBeanFactory` 接口的 `registerCustomEditor()` 方法。另一种略为方便的机制是使用一个被称为 `CustomEditorConfigurer` 的特殊的bean factory后处理器(*post-processor*)。虽然bean factory后处理器可以与 `BeanFactory` 实现一起使用，但是因为 `CustomEditorConfigurer` 有一个嵌套属性设置过程，所以强烈推荐它与 `ApplicationContext` 一起使用，这样就可以采用与其他bean类似的方式来部署它，并自动检测和应用。

请注意所有的bean工厂和应用上下文都会自动地使用一些内置属性编辑器，这些编辑器通过一个被称为 `BeanWrapper` 的接口来处理属性转换。 `BeanWrapper` 注册的那些标准属性编辑器已经列在上一部分。 此外，针对特定的应用程序上下文类型， `ApplicationContext` 会用适当的方法覆盖或添加一些额外的编辑器来处理资源查找。

标准的JavaBeans `PropertyEditor` 实例用于将字符串表示的属性值转换成实际的复杂类型属性。 `CustomEditorConfigurer` ，一个bean factory后处理器，可以为添加额外的 `PropertyEditor` 到 `ApplicationContext` 提供便利支持。

考虑一个用户类 `ExoticType` 和另外一个需要将 `ExoticType` 设为属性的类 `DependsOnExoticType` ：

```
package example;

public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

当东西都被正确设置时，我们希望能够分配字符串给type属性，而 `PropertyEditor` 会在背后将其转换成实际的 `ExoticType` 实例：

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

`PropertyEditor` 实现可能与此类似：

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

最后，我们使用 `CustomEditorConfigurer` 将一个新的 `PropertyEditor` 注册到 `ApplicationContext` ，那么在需要的时候就能够使用它：

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>
```

使用PropertyEditorRegistrar

另一种将属性编辑器注册到Spring容器的机制是创建和使用一个 `PropertyEditorRegistrar` 。当你需要在几个不同场景里使用同一组属性编辑器，这个接口会特别有用：编写一个相应的registrar并在每个用例里重用。 `PropertyEditorRegistrar` 与一个被称为 `PropertyEditorRegistry` 的接口配合工作，后者被Spring的 `BeanWrapper` (以及 `DataBinder`)实现。当与 `CustomEditorConfigurer` 配合使用的时候， `PropertyEditorRegistrar` 特别方便(这里有介绍)，因为前者暴露了一个方法 `setPropertyEditorRegistrars(..)` ：以这种方式添加到 `CustomEditorConfigurerd` 的 `PropertyEditorRegistrar` 可以很容易地在 `DataBinder` 和Spring MVC `Controllers` 之间共享。另外，它避免了在自定义编辑器上的同步需求：一个 `PropertyEditorRegistrar` 可以为每一次bean创建尝试创建新的 `PropertyEditor` 实例。

使用 `PropertyEditorRegistrar` 可能最好还是以一个例子来说明。首先，你需要创建你自己的 `PropertyEditorRegistrar` 实现：

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

也可以查看 `org.springframework.beans.support.ResourceEditorRegistrar` 当作一个 `PropertyEditorRegistrar` 实现的示例。注意在它的 `registerCustomEditors(..)` 方法实现里是如何为每个属性编辑器创建新的实例的。

接着我们配置了一个 `CustomEditorConfigurerd` 并将我们的 `CustomPropertyEditorRegistrar` 注入其中：

```
<bean class="org.springframework.beans.factory.config.CustomEdit
orConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"   class="com.foo.editor
s.spring.CustomPropertyEditorRegistrar"/>
```

最后，有点偏离本章的重点，针对你们之中使用Spring's MVC web framework的那些人，使用 `PropertyEditorRegistrar` 与数据绑定的 `Controller` (比如 `SimpleFormController`)配合使用会非常方便。下面是一个在 `initBinder(..)` 方法的实现里使用 `PropertyEditorRegistrar` 的例子：

```
public final class RegisterUserController extends SimpleFormCont
roller {

    private final PropertyEditorRegistrar customPropertyEditorRe
gistrar;

    public RegisterUserController(PropertyEditorRegistrar proper
tyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegis
trar;
    }

    protected void initBinder(HttpServletRequest request,
            ServletRequestDataBinder binder) throws Exception {
        this.customPropertyEditorRegistrar.registerCustomEditors
(binder);
    }

    // other methods to do with registering a User
}
```

这种 `PropertyEditor` 注册的风格可以导致简洁的代码(`initBinder(..)` 的实现仅仅只有一行！)，同时也允许将通用的 `PropertyEditor` 注册代码封装到一个类里然后根据需要在尽可能多的 `Controller` 之间共享。

5.5 Spring类型转换

Spring 3引入了 `core.convert` 包来提供一个一般类型的转换系统。这个系统定义了实现类型转换逻辑的服务提供接口(SPI)以及在运行时执行类型转换的API。在Spring容器内，这个系统可以当作是PropertyEditor的替代选择，用于将外部bean的属性值字符串转换成所需的属性类型。这个公共的API也可以在你的应用程序中任何需要类型转换的地方使用。

5.5.1 Converter SPI

实现类型转换逻辑的SPI是简单并且强类型的：

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);

}
```

要创建属于你自己的转换器，只需要简单的实现以上接口即可。泛型参数S表示你想要进行转换的源类型，而泛型参数T表示你想要转换的目标类型。如果一个包含S类型元素的集合或数组需要转换为一个包含T类型的数组或集合，那么这个转换器也可以被透明地应用，前提是已经注册了一个委托数组或集合的转换器(默认情况下会是 `DefaultConversionService` 处理)。

对每次方法 `convert(S)` 的调用，source参数值必须确保不为空。如果转换失败，你的转换器可以抛出任何非受检异常(*unchecked exception*)；具体来说，为了报告一个非法的source参数值，应该抛出一个 `IllegalArgumentException` 。还有要注意确保你的 `Converter` 实现必须是线程安全的。

为方便起见， `core.convert.support` 包已经提供了一些转换器实现，这些实现包括了从字符串到数字以及其他常见类型的转换。考虑将 `StringToInteger` 作为一个典型的 `Converter` 实现示例：

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer
> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }

}
```

5.5.2 ConverterFactory

当你需要集中整个类层次结构的转换逻辑时，例如，碰到将String转换到java.lang.Enum对象的时候，请实现 `ConverterFactory` ：

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetTy
pe);

}
```

泛型参数S表示你想要转换的源类型，泛型参数R表示你可以转换的那些范围内的类型的基类。然后实现getConverter(Class)，其中T就是R的一个子类。

考虑将 `StringToEnum` 作为ConverterFactory的一个示例：

```
package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFac
tory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Cl
ass<T> targetType) {
        return new StringToEnumConverter(targetType);
    }

    private final class StringToEnumConverter<T extends Enum> im
plements Converter<String, T> {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim()
);
        }
    }
}
```

5.5.3 GenericConverter

当你需要一个复杂的转换器实现时，请考虑GenericConverter接口。GenericConverter具备更加灵活但是不太强的类型签名，以支持在多种源类型和目标类型之间的转换。此外，当实现你的转换逻辑时，GenericConverter还可以使源字段和目标字段的上下文对你可用，这样的上下文允许类型转换由字段上的注解或者字段声明中的泛型信息来驱动。

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}
```

要实现一个GenericConverter，getConvertibleTypes()方法要返回支持的源-目标类型对，然后实现convert(Object,TypeDescriptor,TypeDescriptor)方法来实现你的转换逻辑。源TypeDescriptor提供了对持有被转换值的源字段的访问，目标TypeDescriptor提供了对设置转换值的目标字段的访问。

一个很好的GenericConverter的示例是一个在Java数组和集合之间进行转换的转换器。这样一个ArrayToCollectionConverter可以通过内省声明了目标集合类型的字段以解析集合元素的类型，这将允许原数组中每个元素可以在集合被设置到目标字段之前转换成集合元素的类型。

> 由于GenericConverter是一个更复杂的SPI接口，所以对基本类型的转换需求优先使用Converter或者ConverterFactory。

ConditionalGenericConverter

有时候你只想要在特定条件成立的情况下 `Converter` 才执行，例如，你可能只想要在目标字段存在特定注解的情况下才执行 `Converter` ，或者你可能只想要在目标类中定义了特定方法，比如 `staticvalueOf` 方法，才执行 `Converter` 。 `ConditionalGenericConverter` 是 `GenericConverter` 和 `ConditionalConveter` 接口的联合，允许你定义这样的自定义匹配条件：

```
public interface ConditionalGenericConverter
        extends GenericConverter, ConditionalConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor ta
rgetType);

}
```

`ConditionalGenericConverter` 的一个很好的例子是一个在持久化实体标识和实体引用之间进行转换的实体转换器。这个实体转换器可能只匹配这样的条件–目标实体类声明了一个静态的查找方法，例如 `findAccount(Long)` ，你将在 `matches(TypeDescriptor,TypeDescriptor)` 方法实现里执行这样的查找方法的检测。

5.5.4 ConversionService API

ConversionService接口定义了运行时执行类型转换的统一API，转换器往往是在这个门面(*facade*)接口背后执行：

```
package org.springframework.core.convert;

public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType)
;

    <T> T convert(Object source, Class<T> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor
 targetType);

    Object convert(Object source, TypeDescriptor sourceType, Typ
eDescriptor targetType);

}
```

大多数ConversionService实现也会实现 `ConverterRegistry` 接口，这个接口提供一个用于注册转换器的服务提供接口(SPI)。在内部，一个ConversionService实现会以委托给注册其中的转换器的方式来执行类型转换逻辑。

`core.convert.support` 包已经提供了一个强大的ConversionService实现， `GenericConversionService` 是适用于大多数环境的通用实现， `ConversionServiceFactory` 以工厂的方式为创建常见的ConversionService配置提供了便利。

5.5.5 配置ConversionService

ConversionService是一个被设计成在应用程序启动时会进行实例化的无状态对象，随后可以在多个线程之间共享。在一个Spring应用程序中，你通常会为每一个Spring容器(或者应用程序上下文ApplicationContext)配置一个ConversionService实例，它会被Spring接收并在框架需要执行一个类型转换时使用。你也可以将这个ConversionService直接注入到你任何的Bean中并直接调用。

> 如果Spring没有注册ConversionService，则会使用原始的基于PropertyEditor的系统。

要向Spring注册默认的ConversionService，可以用 `conversionService` 作为id来添加如下的bean定义：

```
<bean id="conversionService"
     class="org.springframework.context.support.ConversionServic
eFactoryBean"/>
```

默认的ConversionService可以在字符串、数字、枚举、映射和其他常见类型之间进行转换。为了使用你自己的自定义转换器来补充或者覆盖默认的转换器，可以设置 `converters` 属性，该属性值可以是Converter、ConverterFactory或者GenericConverter之中任何一个的接口实现。

```
<bean id="conversionService"
        class="org.springframework.context.support.ConversionSer
viceFactoryBean">
    <property name="converters">
        <set>
            <bean class="example.MyCustomConverter"/>
        </set>
    </property>
</bean>
```

在一个Spring MVC应用程序中使用ConversionService也是比较常见的，可以去看Spring MVC章节的Section 18.16.3 “Conversion and Formatting”。

在某些情况下，你可能希望在转换期间应用格式化，可以看5.6.3 “FormatterRegistry SPI”获取使用 `FormattingConversionServiceFactoryBean` 的细节。

5.5.6 编程方式使用ConversionService

要以编程方式使用ConversionService，你只需要像处理其他bean一样注入一个引用即可：

```
@Service
public class MyService {

    @Autowired
    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

对大多数用例来说， `convert` 方法指定了可以使用的目标类型，但是它不适用于更复杂的类型比如参数化元素的集合。例如，如果你想要以编程方式将一个 `Integer` 的 `List` 转换成一个 `String` 的 `List` ，就需要为原类型和目标类型提供一个正式的定义。

幸运的是， `TypeDescriptor` 提供了多种选项使事情变得简单：

```
DefaultConversionService cs = new DefaultConversionService();


List<Integer> input = ....
cs.convert(input,
    TypeDescriptor.forObject(input), // List<Integer> type descr
iptor
    TypeDescriptor.collection(List.class, TypeDescriptor.valueOf
(String.class)));
```

注意 `DefaultConversionService` 会自动注册对大部分环境都适用的转换器，这其中包括了集合转换器、标量转换器还有基本的 `Object` 到 `String` 的转换器。可以通过调用 `DefaultConversionService` 类上的静态方法 `addDefaultConverters` 来向任意的 `ConverterRegistry` 注册相同的转换器。

因为值类型的转换器可以被数组和集合重用，所以假设标准集合处理是恰当的，就没有必要创建将一个 `S` 的 `Collection` 转换成一个 `T` 的 `Collection` 的特定转换器。

5.6 Spring字段格式化

如上一节所述，`core.convert`包是一个通用类型转换系统，它提供了统一的ConversionService API以及强类型的Converter SPI用于实现将一种类型转换成另外一种的转换逻辑。Spring容器使用这个系统来绑定bean属性值，此外，Spring表达式语言(SpEL)和DataBinder也都使用这个系统来绑定字段值。举个例子，当SpEL需要将`Short`强制转换成`Long`来完成一次`expression.setValue(Object bean, Object value)`尝试时，core.convert系统就会执行这个强制转换。

现在让我们考虑一个典型的客户端环境如web或桌面应用程序的类型转换要求，在这样的环境里，你通常会经历将字符串进行转换以支持客户端回传的过程以及转换回字符串以支持视图渲染的过程。此外，你经常需要对字符串值进行本地化。更通用的*core.convert*包中的Converter SPI不直接解决这种格式化要求。Spring 3为此引入了一个方便的Formatter SPI来直接解决这些问题，这个接口为客户端环境提供一种简单强大并且替代PropertyEditor的方案。

一般来说，当你需要实现通用的类型转换逻辑时请使用Converter SPI，例如，在java.util.Date和java.lang.Long之间进行转换。当你在一个客户端环境(比如web应用程序)工作并且需要解析和打印本地化的字段值时，请使用Formatter SPI。ConversionService接口为这两者提供了一套统一的类型转换API。

5.6.1 Formatter SPI

Formatter SPI实现字段格式化逻辑是简单并且强类型的：

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

Formatter接口扩展了Printer和Parser这两个基础接口：

```
public interface Printer<T> {
    String print(T fieldValue, Locale locale);
}
```

```
import java.text.ParseException;

public interface Parser<T> {
    T parse(String clientValue, Locale locale) throws ParseExcep
tion;
}
```

要创建你自己的格式化器，只需要实现上面的Formatter接口。泛型参数T代表你想要格式化的对象的类型，例如， `java.util.Date` 。实现 `print()` 操作可以将类型T的实例按客户端区域设置的显示方式打印出来。实现 `parse()` 操作可以从依据客户端区域设置返回的格式化表示中解析出类型T的实例。如果解析尝试失败，你的格式化器应该抛出一个ParseException或者IllegalArgumentException。请注意确保你的格式化器实现是线程安全的。

为方便起见， `format` 子包中已经提供了一些格式化器实现。 `number` 包提供了 `NumberFormatter` 、 `CurrencyFormatter` 和 `PercentFormatter` ，它们通过使用 `java.text.NumberFormat` 来格式化 `java.lang.Number` 对象。 `datetime` 包提供了 `DateFormatter` ，其通过使用 `java.text.DateFormat` 来格式化 `java.util.Date` 。 `datetime.joda` 包基于Joda Time library提供了全面的日期时间格式化支持。

考虑将 `DateFormatter` 作为 `Formatter` 实现的一个例子：

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws Pa
rseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.patter
n, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }

}
```

Spring团队欢迎社区驱动的 `Formatter` 贡献，可以登陆网站jira.spring.io了解如何参与贡献。

5.6.2 注解驱动的格式化

如你所见，字段格式化可以通过字段类型或者注解进行配置，要将一个注解绑定到一个格式化器，可以实现AnnotationFormatterFactory：

```
package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation
> {

    Set<Class<?>> getFieldTypes();

    Printer<?> getPrinter(A annotation, Class<?> fieldType);

    Parser<?> getParser(A annotation, Class<?> fieldType);

}
```

泛型参数A代表你想要关联格式化逻辑的字段注解类型，例如 `org.springframework.format.annotation.DateTimeFormat` 。让 `getFieldTypes()` 方法返回可能使用注解的字段类型，让 `getPrinter()` 方法返回一个可以打印被注解字段的值的打印机(Printer)，让 `getParser()` 方法返回一个可以解析被注解字段的客户端值的解析器(Parser)。

下面这个AnnotationFormatterFactory实现的示例把@NumberFormat注解绑定到一个格式化器，此注解允许指定数字样式或模式：

```
public final class NumberFormatAnnotationFormatterFactory
        implements AnnotationFormatterFactory<NumberFormat> {

    public Set<Class<?>> getFieldTypes() {
        return new HashSet<Class<?>>(asList(new Class<?>[] {
            Short.class, Integer.class, Long.class, Float.class,
            Double.class, BigDecimal.class, BigInteger.class }))
;
    }

    public Printer<Number> getPrinter(NumberFormat annotation, C
lass<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    public Parser<Number> getParser(NumberFormat annotation, Cla
ss<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    private Formatter<Number> configureFormatterFrom(NumberForma
t annotation,
            Class<?> fieldType) {
        if (!annotation.pattern().isEmpty()) {
            return new NumberFormatter(annotation.pattern());
        } else {
            Style style = annotation.style();
            if (style == Style.PERCENT) {
                return new PercentFormatter();
            } else if (style == Style.CURRENCY) {
                return new CurrencyFormatter();
            } else {
                return new NumberFormatter();
            }
        }
    }
}
```

要触发格式化，只需要使用@NumberFormat对字段进行注解：

```
public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;

}
```

Format Annotation API

`org.springframework.format.annotation` 包中存在一套可移植(portable)的格式化注解API。请使用@NumberFormat格式化java.lang.Number字段，使用@DateTimeFormat格式化java.util.Date、java.util.Calendar、*java.util.Long(*注：此处可能是原文错误，应为*java.lang.Long)*或者Joda Time字段。

下面这个例子使用@DateTimeFormat将java.util.Date格式化为ISO时间(yyyy-MM-dd)

```
public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;

}
```

5.6.3 FormatterRegistry SPI

FormatterRegistry是一个用于注册格式化器和转换器的服务提供接口(SPI)。 `FormattingConversionService` 是一个适用于大多数环境的FormatterRegistry实现，可以以编程方式或利用 `FormattingConversionServiceFactoryBean` 声明成Spring bean的方式来进行配置。由于它也实现了 `ConversionService` ，所以可以直接配置它与Spring的DataBinder以及Spring表达式语言(SpEL)一起使用。

请查看下面的FormatterRegistry SPI：

```
package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

    void addFormatterForFieldType(Class<?> fieldType, Printer<?>
 printer, Parser<?> parser);

    void addFormatterForFieldType(Class<?> fieldType, Formatter<
?> formatter);

    void addFormatterForFieldType(Formatter<?> formatter);

    void addFormatterForAnnotation(AnnotationFormatterFactory<?,
 ?> factory);

}
```

如上所示，格式化器可以通过字段类型或者注解进行注册。

FormatterRegistry SPI允许你集中地配置格式化规则，而不是在你的控制器之间重复这样的配置。例如，你可能要强制所有的时间字段以某种方式被格式化，或者是带有特定注解的字段以某种方式被格式化。通过一个共享的FormatterRegistry，你可以只定义这些规则一次，而在需要格式化的时候应用它们。

5.6.4 FormatterRegistrar SPI

FormatterRegistrar是一个通过FormatterRegistry注册格式化器和转换器的服务提供接口(SPI)：

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);

}
```

当要为一个给定的格式化类别(比如时间格式化)注册多个关联的转换器和格式化器时，FormatterRegistrar会非常有用。

下一部分提供了更多关于转换器和格式化器注册的信息。

5.6.5 在Spring MVC中配置格式化

请查看Spring MVC章节的Section 18.16.3 “Conversion and Formatting”。

5.7 配置一个全局的日期&时间格式

默认情况下，未被 `@DateTimeFormat` 注解的日期和时间字段会使用 `DateFormat.SHORT` 风格从字符串转换。如果你愿意，你可以定义你自己的全局格式来改变这种默认行为。

你将需要确保Spring不会注册默认的格式化器，取而代之的是你应该手动注册所有的格式化器。请根据你是否依赖Joda Time库来确定是使用 `org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar` 类还是 `org.springframework.format.datetime.DateFormatterRegistrar` 类。

例如，下面的Java配置会注册一个全局的'yyyyMMdd'格式，这个例子不依赖于Joda Time库：

```
@Configuration
public class AppConfig {

    @Bean
    public FormattingConversionService conversionService() {

        // Use the DefaultFormattingConversionService but do not
 register defaults
        DefaultFormattingConversionService conversionService = n
ew DefaultFormattingConversionService(false);

        // Ensure @NumberFormat is still supported
        conversionService.addFormatterForFieldAnnotation(new Num
berFormatAnnotationFormatterFactory());

        // Register date conversion with a specific global forma
t
        DateFormatterRegistrar registrar = new DateFormatterRegi
strar();
        registrar.setFormatter(new DateFormatter("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        return conversionService;
    }
}
```

如果你更喜欢基于XML的配置，你可以使用一个 `FormattingConversionServiceFactoryBean` ，这是同一个例子，但这次使用了Joda Time：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd>

    <bean id="conversionService" class="org.springframework.form
at.support.FormattingConversionServiceFactoryBean">
        <property name="registerDefaultFormatters" value="false"
 />
        <property name="formatters">
            <set>
                <bean class="org.springframework.format.number.N
umberFormatAnnotationFormatterFactory" />
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.springframework.format.datetime
.joda.JodaTimeFormatterRegistrar">
                    <property name="dateFormatter">
                        <bean class="org.springframework.format.
datetime.joda.DateTimeFormatterFactoryBean">
                            <property name="pattern" value="yyyy
MMdd"/>
                        </bean>
                    </property>
                </bean>
            </set>
        </property>
    </bean>
</beans>
```

> Joda Time提供了不同的类型来表示 `date` 、 `time` 和 `date-time` 的值， `JodaTimeFormatterRegistrar` 中的 `dateFormatter` 、 `timeFormatter` 和 `dateTimeFormatter` 属性应该为每种类型配置不同的格式。 `DateTimeFormatterFactoryBean` 提供了一种方便的方式来创建格式化器。在

如果你在使用Spring MVC，请记住要明确配置所使用的转换服务。针对基于 `@Configuration` 的Java配置方式这意味着要继承 `WebMvcConfigurationSupport` 并且覆盖 `mvcConversionService()` 方法。针对XML的方式，你应该使用 `mvc:annotation-drive` 元素的 `'conversion-service'` 属性。更多细节请看Section 18.16.3 “Conversion and Formatting”。

5.8 Spring验证

Spring 3对验证支持引入了几个增强功能。首先，现在全面支持JSR-303 Bean Validation API；其次，当采用编程方式时，Spring的DataBinder现在不仅可以绑定对象还能够验证它们；最后，Spring MVC现在已经支持声明式地验证 `@Controller` 的输入。

5.8.1 JSR-303 Bean Validation API概述

JSR-303对Java平台的验证约束声明和元数据进行了标准化定义。使用此API，你可以用声明性的验证约束对领域模型的属性进行注解，并在运行时强制执行它们。现在已经有一些内置的约束供你使用，当然你也可以定义你自己的自定义约束。

为了说明这一点，考虑一个拥有两个属性的简单的PersonForm模型：

```
public class PersonForm {
    private String name;
    private int age;
}
```

JSR-303允许你针对这些属性定义声明性的验证约束：

```
public class PersonForm {

    @NotNull
    @Size(max=64)
    private String name;

    @Min(0)
    private int age;

}
```

当此类的一个实例被实现JSR-303规范的验证器进行校验的时候，这些约束就会被强制执行。

有关JSR-303/JSR-349的一般信息，可以访问网站Bean Validation website去查看。有关默认参考实现的具体功能的信息，可以参考网站Hibernate Validator的文档。想要了解如何将Bean验证器提供程序设置为Spring bean，请继续保持阅读。

5.8.2 配置Bean验证器提供程序

Spring提供了对Bean Validation API的全面支持，这包括将实现JSR-303/JSR-349规范的Bean验证提供程序引导为Spring Bean的方便支持。这样就允许在应用程序任何需要验证的地方注入 `javax.validation.ValidatorFactory` 或者 `javax.validation.Validator` 。

把 `LocalValidatorFactoryBean` 当作Spring bean来配置成默认的验证器：

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalVa
lidatorFactoryBean"/>
```

以上的基本配置会触发Bean Validation使用它默认的引导机制来进行初始化。作为实现JSR-303/JSR-349规范的提供程序，如Hibernate Validator，可以存在于类路径以使它能被自动检测到。

注入验证器

`LocalValidatorFactoryBean` 实现了 `javax.validation.ValidatorFactory` 和 `javax.validation.Validator` 这两个接口，以及Spring的 `org.springframework.validation.Validator` 接口，你可以将这些接口当中的任意一个注入到需要调用验证逻辑的Bean里。

如果你喜欢直接使用Bean Validtion API，那么就注入 `javax.validation.Validator` 的引用：

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
```

如果你的Bean需要Spring Validation API，那么就注入 `org.springframework.validation.Validator` 的引用：

```
import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;

}
```

配置自定义约束

每一个Bean验证约束由两部分组成，第一部分是声明了约束和其可配置属性的 `@Constraint` 注解，第二部分是实现约束行为的 `javax.validation.ConstraintValidator` 接口实现。为了将声明与实现关联起来，每个 `@Constraint` 注解会引用一个相应的验证约束的实现类。在运行期间， `ConstraintValidatorFactory` 会在你的领域模型遇到约束注解的情况下实例化被引用到的实现。

默认情况下， `LocalValidatorFactoryBean` 会配置一个 `SpringConstraintValidatorFactory` ，其使用Spring来创建约束验证器实例。这允许你的自定义约束验证器可以像其他Spring bean一样从依赖注入中受益。

下面显示了一个自定义的 `@Constraint` 声明的例子，紧跟着是一个关联的 `ConstraintValidator` 实现，其使用Spring进行依赖注入：

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

```
import javax.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidato
r {

    @Autowired;
    private Foo aDependency;

    ...
}
```

如你所见，一个约束验证器实现可以像其他Spring bean一样使用@Autowired注解来自动装配它的依赖。

Spring驱动的方法验证

被Bean Validation 1.1以及作为Hibernate Validator 4.3中的自定义扩展所支持的方法验证功能可以通过配置 `MethodValidationPostProcessor` 的bean定义集成到Spring的上下文中：

```
<bean class="org.springframework.validation.beanvalidation.Metho
dValidationPostProcessor"/>
```

为了符合Spring驱动的方法验证，需要对所有目标类用Spring的 `@Validated` 注解进行注解，且有选择地对其声明验证组，这样才可以使用。请查阅 `MethodValidationPostProcessor` 的java文档来了解针对Hibernate Validator和Bean Validation 1.1提供程序的设置细节。

附加配置选项

对于大多数情况，默认的 `LocalValidatorFactoryBean` 配置应该足够。有许多配置选项来处理从消息插补到遍历解析的各种Bean验证结构。请查看 `LocalValidatorFactoryBean` 的java文档来获取关于这些选项的更多信息。

5.8.3 配置DataBinder

从Spring 3开始，DataBinder的实例可以配置一个验证器。一旦配置完成，那么可以通过调用 `binder.validate()` 来调用验证器，任何的验证错误都会自动添加到DataBinder的绑定结果(BindingResult)。

当以编程方式处理DataBinder时，可以在绑定目标对象之后调用验证逻辑：

```
Foo target = new Foo();
DataBinder binder = new DataBinder(target);
binder.setValidator(new FooValidator());

// bind to the target object
binder.bind(propertyValues);

// validate the target object
binder.validate();

// get BindingResult that includes any validation errors
BindingResult results = binder.getBindingResult();
```

通过 `dataBinder.addValidators` 和 `dataBinder.replaceValidators` ，一个DataBinder也可以配置多个 `Validator` 实例。当需要将全局配置的Bean验证与一个DataBinder实例上局部配置的Spring `Validator` 结合时，这一点是非常有用的。

5.8.4 Spring MVC 3 验证

请查看Spring MVC章节的Section 18.16.4 “Validation”。

6. Spring 表达式语言

6.1 介绍

Spring Expression Language（简称SpEL）是一种功能强大的表达式语言、用于在运行时查询和操作对象图；语法上类似于Unified EL，但提供了更多的特性，特别是方法调用和基本字符串模板函数。

虽然目前已经有许多其他的Java表达式语言，例如OGNL，MVEL和Jboss EL，SpEL的诞生是为了给Spring社区提供一种能够与Spring生态系统所有产品无缝对接，能提供一站式支持的表达式语言。它的语言特性由Spring生态系统的实际项目需求驱动而来，比如基于eclipse的Spring Tool Suite（Spring开发工具集）中的代码补全工具需求。尽管如此、SpEL本身基于一套与具体实现技术无关的API，在需要的时候允许其他的表达式语言实现集成进来。

尽管SpEL在Spring产品中是作为表达式求值的核心基础模块，本身可以脱离Spring独立使用。为了体现它的独立性，本章节中的许多例子都将SpEL作为独立的表达式语言来使用。不过这样就需要每次都先创建一些基础框架类如解析器，而对于大多数Spring用户来说并不需要去关注这些基础框架类，仅仅只需要写相应的字符串求值表达式即可。一个典型的例子就是把SpEL集成进XML bean配置或者基于注解的Bean定义声明中（详见章节：Expression support for defining bean definitions）

本章节包含SpEL的语言特性，它的API及语法。很多地方用到了Inventor类及相关的Society类作为表达式求值的操作例子对象，这几个类的定义及操作它们的数据都列在本章的末尾.

6.2 功能特性

SpEL支持以下的一些特性：

- 字符表达式
- 布尔和关系操作符
- 正则表达式
- 类表达式
- 访问properties，arrays，lists，maps等集合
- 方法调用
- 关系操作符
- 赋值
- 调用构造器
- Bean对象引用
- 创建数组
- 内联lists
- 内联maps
- 三元操作符
- 变量
- 用户自定义函数
- 集合投影
- 集合选择
- 模板表达式

6.3 使用SpEL的接口进行表达式求值

本节介绍SpEL接口及其表达式语言的简单使用方法。完整的语言文档见：Language Reference

下面代码介绍了使用SpEL API来解析字符串表达式'Hello World'的示例

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

最常用的SpEL类和接口都放在包org.springframework.expression及其子包和spel.support下

接口ExpressionParser用来解析一个字符串表达式。在这个例子中字符串表达式即用单引号括起来的字符串.接口Expression用于对上面定义的字符串表达式求值。调用parser.parseExpression和exp.getValue分别可能抛出ParseException和EvaluationException。

SpEL支持一系列广泛的特性，例如方法调用，访问属性，调用构造函数等。

下面举一个方法调用的例子，在String文本后面调用concat方法。

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!
')");
String message = (String) exp.getValue();
```

message的值现在就是 'Hello World!'.

接下去是一个访问JavaBean属性的例子，String类的Bytes属性通过以下的方法调用

```
// 调用'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes");
byte[] bytes = (byte[]) exp.getValue();
```

SpEL同时也支持级联属性调用、和标准的prop1.prop2.prop3方式是一样的；同样属性值设置也是类似的方式.

公共方法也可以被访问到：

```
ExpressionParser parser = new SpelExpressionParser();

// 调用 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.len
gth");
int length = (Integer) exp.getValue();
```

除了使用字符串表达式、也可以调用String的构造函数：

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world
').toUpperCase()");
String message = exp.getValue(String.class);
```

针对泛型方法的使用，例如：public <T> T getValue(Class<T> desiredResultType) 使用这样的方法不需要将表达式的值转换成具体的结果类型。如果具体的值类型或者使用类型转换器都无法转成对应的类型、会抛EvaluationException的异常

SpEL中更常见的用途是提供一个针对特定对象实例（叫做根对象）求值的表达式字符串。使用方法有两种，
具体用哪一种要看每次调用表达式求值时相应的对象实例是否每次都会变化。接下来我们分别举两个例子说明，
第一个例子我们要做的是从Inventor类的实例中解析name属性：

```
// 创建并设置一个calendar实例
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// 构造器参数有： name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serb
ian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");

EvaluationContext context = new StandardEvaluationContext(tesla)
;
String name = (String) exp.getValue(context);
```

最后一行，字符串变量name将会被设置成”Nikola Tesla”. 通过StandardEvaluationContext类你能指定哪个对象的”name”会被求值这种机制用于当根对象不会被改变的场景，在求值上下文中只会被设置一次。相反，如果根对象会经常改变，则需要在每次调用getValue的时候被设置，就像如下的示例代码：

```
// 创建并设置一个calendar实例
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// 构造器参数有： name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serb
ian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
String name = (String) exp.getValue(tesla);
```

在上面这个例子中inventor类实例tesla直接在getValue中设置，表达式解析器底层框架会自动创建和管理一个默认的求值上下文-不需要被显式声明

StandardEvaluationContext创建相对比较耗资源，在重复多次使用的场景下内部会缓存部分中间状态加快后续的表达式求值效率。因此建议在使用过程中尽可能被缓存和重用，而不是每次在表达式求值时都重新创建一个对象。

在很多使用场景下理想的方式是事先配置好求值上下文，但是在实际调用中仍然可以给getValue设置一个不同的根对象。getValue允许在同一个调用中同时指定两者，在这种场景下运行时传入的根对象会覆盖在求值上下文中事先指定的根对象。

> Note：在单独使用SpEL时需要创建解析器、解析表达式、以及求值上下文和对应的根对象。但是在实际使用过程中、更常用的使用方式是只需要在配置文件里面配置SpEL字符串表达式即可，例如针对Spring Bean或者Spring Web Flow的定义。在这种场景下解析器，求值上下文，根对象和任何事先定义的变量都会被容器默认创建好，用户除了写表达式不需要做任何其他事情。

作为最后一个介绍性的例子，我们沿用上面的例子写一个使用布尔操作符的求值例子

```
Expression exp = parser.parseExpression("name == 'Nikola Tesla'"
);
boolean result = exp.getValue(context, Boolean.class); // 求值结
果是True
```

6.3.1 EvaluationContext接口

EvaluationContext接口在求值表达式中需要解析属性，方法，字段的值以及类型转换中会用到。其默认实现类StandardEvaluationContext使用反射机制来操作对象。为获得更好的性能缓存了 `java.lang.reflect.Method` , `java.lang.reflect.Field` ,和 `java.lang.reflect.Constructor实例` 。

StandardEvaluationContext中你可以使用setRootObject()方法显式设置根对象，或通过构造器直接传入根对象。你还可以通过调用setVariable()和registerFunction()方法指定在表达式中用到的变量和函数。变量和函数的使用在语言参考的 Variables和Functions两章节有详细说明。使用StandardEvaluationContext你还可以注册自定义的构造器解析器（ConstructorResolvers）,方法解析器（MethodResolvers）,和属性存取器（PropertyAccessor）来扩展SpEL如何计算表达式，详见具体类的JavaDoc文档。

类型转换

SpEL默认使用Spring核心代码中的conversion service来做类型转换（org.springframework.core.convert.ConversionService）。这个类本身内置了很多常用的转换器，同时也可以扩展使用自定义的类型转换器。另外一个核心功能是它可以识别泛型。这意味着当在表达式中使用泛型类型时、SpEL会确保任何处理的对象的类型正确性。

实际应用中这意味着什么？这里举一个例子、拿赋值来说，比如使用setValue来设置List属性。属性的类型实际上是List<Boolean>，SpEL可以识别List中的元素类型并转换成Boolean类型。下面是示例代码：

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();

simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new StandardEvaluation
Context(simple);

// false is passed in here as a string. SpEL and the conversion
service will
// correctly recognize that it needs to be a Boolean and convert
 it
parser.parseExpression("booleanList[0]").setValue(simpleContext,
 "false");

// b will be false
Boolean b = simple.booleanList.get(0);
```

6.3.2 解析器配置

可以通过使用一个解析器配置对象(org.springframework.expression.spel.SpelParserConfiguration)来配置SpEL表达式解析器。这个配置对象可以控制一些表达式组件的行为。例如：数组或者集合元

素查找的时候如果当前位置对应的对象是Null，可以通过事先配置来自动创建元素。这个在表达式是由多次属性链式引用的时候比较重要。如果设置的数组或者List位置越界时可以自动增加数组或者List长度来兼容.

```
class Demo {
    public List<String> list;
}

// Turn on:
// - auto null reference initialization
// - auto collection growing
SpelParserConfiguration config = new SpelParserConfiguration(tru
e,true);

ExpressionParser parser = new SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = new Demo();

Object o = expression.getValue(demo);

// demo.list will now be a real collection of 4 entries
// Each entry is a new empty String
```

SpEL表达式编译器的行为也可以通过配置来实现

6.3.3 SpEL编译

Spring4.1 包括了一个基本的表达式编译器。表达式求值过程中往往可以利用动态解释功能来提供很多灵活性，但是却达不到理想的性能。当表达式不是被经常使用时是可以接受的，但是当被其他组件像Spring Integration使用时，性能往往更重要，这时动态性没有太多的需求。

新的SpEL编译器针对这样的需求进行了定制。编译器会在求值时同时创建一个真实的Java类包含表达式的行为，利用这种方式来达到更快的表达式求值速度。在编译过程中、编译器刚开始不知道表达式具体的类型、但它会在表达式求值过程中收集相应的信息来确定最终的类型。例如，编译器无法仅仅从表达式自身来分析出某

个属性引用的类型，但是在首次进行解释求值时就可以确定了。当然，如果不同的表达式元素类型后续发生变化、基于这种信息来编译结果就不准确了。因此编译最适合的场景是表达式在多次求值过程中类型信息不会变化。

例如一个简单的表达式如下：

```
someArray[0].someProperty.someOtherProperty < 0.1
```

这个表达式包含数组访问，属性引用和数值运算，其性能开销不可忽视。
在一次50000次循环的性能基准评测中，如果只用解析器需要耗时75毫秒，但如果使用已编译的版本则只需要3毫秒

编译器配置

编译器模式不是默认打开的，有两种方法可以打开。一种是前面已经提到过的解析器配置的时候，另一种是当SpEL集成到其他组件时通过设置系统属性的方式打开。本节会同时介绍两种方式.

首先比较重要的是编译器本身有几种操作模式，都定义在枚举类(org.springframework.expression.spel.SpelCompilerMode)中。所有的模式如下：

OFF-编译器关闭；默认是关闭的
IMMEDIATE-即时生效模式，表达式会尽快的被编译。基本是在第一次求值后马上就会执行。如果编译表达式出错（往往都是因为上面提到的类型发生改变的情况）则表达式求值的调用点会抛出异常。
MIXED-混合模式，在混合模式中表达式会自动在解释器模式和编译器模式之间切换。在发生了几次解释处理后会切换到编译模式，如果编译模式哪里出错了（像上面提到的类型发生变化）则表达式会自动切换回解释器模式。过一段时间如果运用正常又会切换回编译模式。基本上像在IMMEDIATE模式下会抛出的那些异常都会被内部处理掉。

即时生效模式之所以会存在是因为混合模式会带来副作用。如果一个已编译的表达式在部分执行成功后发生错误的话，有可能已经导致系统的状态发生变化。这种场景下调用方并不希望表达式在解释模式下重新执行一遍、因为这意味着部分表达式会被执行两遍。

在选择了一个模式后，使用SpelParserConfiguration 来配置解释器：

```
SpelParserConfiguration config = new SpelParserConfiguration(Spe
lCompilerMode.IMMEDIATE,
    this.getClass().getClassLoader());

SpelExpressionParser parser = new SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = new MyMessage();

Object payload = expr.getValue(message);
```

指定编译器类型时也可以同时指定类加载器(不指定传入Null也是允许的)。已编译的表达式将会被定义在所有被创建的子类加载器中。比较重要的一点是一旦指定了一个类加载器、需要确保表达式求值过程中所有涉及的类型对它都是可见的。如果没有明确指定则会使用缺省的类加载器（一般是当前正在执行表达式求值的线程所关联的上下文类加载器）

第二种方法是当SpEL内嵌到其他组件时仅通过配置对象不太容易配置实现的时候使用。在这种场景下往往使用系统属性来设置。属性 spring.expression.compiler.mode可以被设置成SpelCompilerMode枚举值的其中一种（off, immediate, 或者 mixed）。

编译器的局限性

在Srping4.1中基础的编译框架就已经有了。但框架还不支持编译所有类型的表达式。最初的设计初衷主要在于先集中优化比较耗性能且又经常使用的表达式。以下类型的表达式目前还不能被编译：

涉及到赋值的表达式
依赖于转换服务的表达式
使用到自定义解析器或者存取器的表达式
使用到选择器或者投影的表达式

更多类型的表达式将来都会被支持编译

6.4 Bean定义时使用表达式

无论XML还是注解类型的Bean定义都可以使用SpEL表达式。在两种方式下定义的表达式语法都是一样的，即：#{ }

6.4.1 XML类型的配置

Bean属性或者构造函数使用表达式的方式如下：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).ra
ndom() * 100.0 }"/>

    <!-- other properties -->
</bean>
```

在下面的例子中systemProperties 事先已被定义好，因此表达式中可以直接使用。注意：在已定义的变量前无需加#

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator
">
    <property name="defaultLocale" value="#{ systemProperties['u
ser.region'] }"/>

    <!-- other properties -->
</bean>
```

你还可以通过Name注入的方式使用其他Bean的属性，例如：

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).ra
ndom() * 100.0 }"/>

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.rand
omNumber }"/>

    <!-- other properties -->
</bean>
```

6.4.2 基于注解的配置

@Value可以在属性字段，方法和构造器变量中使用，指定一个默认值。

下面的例子中给属性字段设置默认值：

```
public static class FieldValueTestBean

    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }

}
```

通过Set方法设置默认值：

```
public static class PropertyValueTestBean

    private String defaultLocale;

    @Value("#{ systemProperties['user.region'] }")
    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }

}
```

使用Autowired注解的方法和构造器也可以使用@Value注解.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;
    private String defaultLocale;

    @Autowired
    public void configure(MovieFinder movieFinder,
            @Value("#{ systemProperties['user.region'] }") Strin
g defaultLocale) {
        this.movieFinder = movieFinder;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

```
public class MovieRecommender {

    private String defaultLocale;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPrefer
enceDao,
            @Value("#{systemProperties['user.country']}") String
 defaultLocale) {
        this.customerPreferenceDao = customerPreferenceDao;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

6.5 语言参考

6.5.1 字符表达式

字符串表达式类型支持strings,数值类型 (int, real, hex),布尔和null.strings值通过单引号引用。如果字符串里面又包含字符串，通过双引号引用。

下面列出了字符串表达式的常用例子。通常它们不会被单独使用，而是结合一个更复杂的表达式一起使用，例如，在逻辑比较运算符中使用表达式：

```
ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"
String helloWorld = (String) parser.parseExpression("'Hello Worl
d'").getValue();

double avogadrosNumber = (Double) parser.parseExpression("6.0221
415E+23").getValue();

// evals to 2147483647
int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").ge
tValue();

boolean trueValue = (Boolean) parser.parseExpression("true").get
Value();

Object nullValue = parser.parseExpression("null").getValue();
```

数字类型支持负数，指数和小数点。默认情况下实数会使用Double.parseDouble()解析。

6.5.2 属性, 数组, 列表, Maps, 索引

属性引用比较简单：只需要用点号(.)标识级联的各个属性值。Inventor类的实例：pupin,和tesla，所用到的数据在Classes used in the examples一节有列出。
下面的表达式示例用来解析Tesla的出生年及Pupin的出生城市。

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year + 19
00").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City
").getValue(context);
```

属性名的第一个字母可以是大小写敏感的。数组和列表的内容可以使用方括号来标记

```
ExpressionParser parser = new SpelExpressionParser();

// Inventions Array
StandardEvaluationContext teslaContext = new StandardEvaluationC
ontext(tesla);

// evaluates to "Induction motor"
String invention = parser.parseExpression("inventions[3]").getVa
lue(
        teslaContext, String.class);

// Members List
StandardEvaluationContext societyContext = new StandardEvaluatio
nContext(ieee);

// evaluates to "Nikola Tesla"
String name = parser.parseExpression("Members[0].Name").getValue
(
        societyContext, String.class);

// List and Array navigation
// evaluates to "Wireless communication"
String invention = parser.parseExpression("Members[0].Inventions
[6]").getValue(
        societyContext, String.class);
```

Maps的值由方括号内指定字符串的Key来标识引用。在下面这个例子中，因为Officers map的Key是string类型，我们可以用过字符串常量指定。

```
// Officer's Dictionary

Inventor pupin = parser.parseExpression("Officers['president']")
.getValue(
        societyContext, Inventor.class);

// evaluates to "Idvor"
String city = parser.parseExpression("Officers['president'].Plac
eOfBirth.City").getValue(
        societyContext, String.class);

// setting values
parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Cou
ntry").setValue(
        societyContext, "Croatia");
```

6.5.3 内联列表

列表（Lists）可以用过大括号{}直接引用

```
// evaluates to a Java list containing the four numbers
List numbers = (List) parser.parseExpression("{1,2,3,4}").getVal
ue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x
','y'}}").getValue(context);
```

{}本身代表一个空list.因为性能的关系，如果列表本身完全由固定的常量值组成，这个时候会创建一个常量列表来代替表达式，而不是每次在求值的时候创建一个新的列表。

6.5.4 内联Maps

Maps也可以直接通过{key:value}标记的方式在表达式中使用

```
// evaluates to a Java map containing the two entries
Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',
dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Niko
la',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValu
e(context);
```

{:} 本身代表一个空的Map。因为性能的原因：如果Map本身包含固定的常量或者其他级联的常量结构（lists或者maps）则一个常量Map会创建来代表表达式，而不是每次求值的时候都创建一个新的Map.Map的Key并不一定需要引号引用、比如上面的例子就没有引用。

6.5.5 创建数组

数组可以使用类似于Java的语法创建，创建时可以事先指定数组的容量大小、这个是可选的。

```
int[] numbers1 = (int[]) parser.parseExpression("new int[4]").ge
tValue(context);

// Array with initializer
int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3
}").getValue(context);

// Multi dimensional array
int[][] numbers3 = (int[][]) parser.parseExpression("new int[4][
5]").getValue(context);
```

在创建多维数组时还不支持事先指定初始化的值。

6.5.6 方法

方法可以使用典型的Java编程语法来调用。方法可以直接在字符串常量上调用。可变参数也是支持的。

```
// string literal, evaluates to "bc"
String c = parser.parseExpression("'abc'.substring(2, 3)").getVa
lue(String.class);

// evaluates to true
boolean isMember = parser.parseExpression("isMember('Mihajlo Pup
in')").getValue(
        societyContext, Boolean.class);
```

6.5.7 运算符

关系运算符

关系运算符；包括==,<>,<,<=,>,>=等标准运算符都是可以直接支持的

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Bo
olean.class);

// evaluates to false
boolean falseValue = parser.parseExpression("2 &amp;lt; -5.0").g
etValue(Boolean.class);

// evaluates to true
boolean trueValue = parser.parseExpression("'black' &amp;lt; 'bl
ock'").getValue(Boolean.class);
```

> Note：>/<运算符和null做比较时遵循一个简单的规则:null代表什么都没有（不代表0）.因此，所有的值总是大于null(X>null总是true)
> 也就是没有一个值会小于什么都没有(x<null总是返回false).尽量不要在数值比较中使用null,而是和0做比较（例如X>0或者X<0）.

除了标准的关系运算符，SpEL还支持instanceof关键字和基于matches操作符的正则表达式。

> 备注：需要注意元数据类型会自动装箱成包装类型，因此1 instanceof T(int)结果是false，1 instanceof T(Integer)的结果是true.

每一个符号操作符也可以通过首字母简写的方式标识。这样可以避免表达式所用的符号在当前表达式所在的文档中存在特殊含义而带来的冲突（比如XML文档的<）. 简写的符号有：lt (<), gt (>), le (?), ge (>=), eq (==), ne (!=), div (/), mod (%), not (!). 这里不区分大小写。

逻辑运算符

支持的逻辑运算符有：and,or,和not.它们的使用方法如下：

```
// -- AND --

// evaluates to false
boolean falseValue = parser.parseExpression("true and false").ge
tValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') and isMember('Miha
jlo Pupin')";
boolean trueValue = parser.parseExpression(expression).getValue(
societyContext, Boolean.class);

// -- OR --

// evaluates to true
boolean trueValue = parser.parseExpression("true or false").getV
alue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') or isMember('Alber
t Einstein')";
boolean trueValue = parser.parseExpression(expression).getValue(
societyContext, Boolean.class);

// -- NOT --

// evaluates to false
boolean falseValue = parser.parseExpression("!true").getValue(Bo
olean.class);

// -- AND and NOT --
String expression = "isMember('Nikola Tesla') and !isMember('Mih
ajlo Pupin')";
boolean falseValue = parser.parseExpression(expression).getValue
(societyContext, Boolean.class);
```

算术运算符

加号运算符可以同时用于数字和字符串。减号，乘法和除法只能用于数字。其他支持的算术运算符有取模(%)和指数(^).遵循标准运算符优先级。下面是一些例子：

```
// Addition
int two = parser.parseExpression("1 + 1").getValue(Integer.class
); // 2

String testString = parser.parseExpression(
        "'test' + ' ' + 'string'").getValue(String.class); // 't
est string'

// Subtraction
int four = parser.parseExpression("1 - -3").getValue(Integer.cla
ss); // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Doub
le.class); // -9000

// Multiplication
int six = parser.parseExpression("-2 * -3").getValue(Integer.cla
ss); // 6

double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getV
alue(Double.class); // 24.0

// Division
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer
.class); // -2

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Do
uble.class); // 1.0

// Modulus
int three = parser.parseExpression("7 % 4").getValue(Integer.cla
ss); // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.c
lass); // 1

// Operator precedence
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(
Integer.class); // -21
```

6.5.8 赋值

属性可以使用赋值运算符设置。可以通过调用setValue设置。但是也可以在getValue方法中设置

```
Inventor inventor = new Inventor();
StandardEvaluationContext inventorContext = new StandardEvaluati
onContext(inventor);

parser.parseExpression("Name").setValue(inventorContext, "Alexan
der Seovic2");

// alternatively

String aleks = parser.parseExpression(
        "Name = 'Alexandar Seovic'").getValue(inventorContext, S
tring.class);
```

6.5.9 类型

T操作符是一个特殊的操作符、可以同于指定java.lang.Class的实例（类型）。静态方法也可以通过这个操作符调用。

StandardEvaluationContext使用TypeLocator来查找类型，其中StandardTypeLocator（这个可以被替换使用其他类）默认对java.lang包里的类型可见。也就是说 T()引用java.lang包里面的类型不需要限定包全名，但是其他类型的引用必须要。

```
Class dateClass = parser.parseExpression("T(java.util.Date)").ge
tValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue
(Class.class);

boolean trueValue = parser.parseExpression(
        "T(java.math.RoundingMode).CEILING &amp;lt; T(java.math.
RoundingMode).FLOOR")
        .getValue(Boolean.class);
```

6.5.10 构造器

构造器可以使用new操作符来调用。除了元数据类型和String（比如int,float等可以直接使用）都需要限定类的全名。

```
nventor einstein = p.parseExpression(
        "new org.spring.samples.spel.inventor.Inventor('Albert E
instein', 'German')")
        .getValue(Inventor.class);

//create new inventor instance within add method of List
p.parseExpression(
        "Members.add(new org.spring.samples.spel.inventor.Invent
or(
            'Albert Einstein', 'German'))").getValue(societyCont
ext);
```

6.5.11 变量

表达式中的变量可以通过语法#变量名使用。变量可以在StandardEvaluationContext中通过方法setVariable设置。

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContex
t(tesla);
context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context);

System.out.println(tesla.getName()) // "Mike Tesla"
```

#this和#root变量

#this变量永远指向当前表达式正在求值的对象（这时不需要限定全名）。变量#root总是指向根上下文对象。#this在表达式不同部分解析过程中可能会改变，但是#root总是指向根

```
// create an array of integers
List<Integer> primes = new ArrayList<Integer>();
primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

// create parser and set variable 'primes' as the array of integ
ers
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContex
t();
context.setVariable("primes",primes);

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
List<Integer> primesGreaterThanTen = (List<Integer>) parser.pars
eExpression(
        "#primes.?[#this>10]").getValue(context);
```

6.5.12 函数

你可以扩展SpEL，在表达式字符串中使用自定义函数。这些自定义函数是通过StandardEvaluationContext的registerFunction来注册的

```
public void registerFunction(String name, Method m)
```

首先定义一个Java方法作为函数的实现、例如下面是一个将字符串反转的方法。

```
public abstract class StringUtils {

    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder();
        for (int i = 0; i &amp;lt; input.length(); i++)
            backwards.append(input.charAt(input.length() - 1 - i
));
        }
        return backwards.toString();
    }
}
```

然后将这个方法注册到求值上下文中就可以应用到表达式字符串中。

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContex
t();

context.registerFunction("reverseString",
    StringUtils.class.getDeclaredMethod("reverseString", new Cla
ss[] { String.class }));

String helloWorldReversed = parser.parseExpression(
    "#reverseString('hello')").getValue(context, String.class);
```

6.5.13 Bean引用

如果求值上下文已设置bean解析器，可以在表达式中使用（@）符合来查找Bean

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContex
t();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"foo") on MyBeanReso
lver during evaluation
Object bean = parser.parseExpression("@foo").getValue(context);
```

如果是访问工厂Bean，bean名字前需要添加前缀(&)符号

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContex
t();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"&amp;amp;foo") on M
yBeanResolver during evaluation
Object bean = parser.parseExpression("&amp;amp;foo").getValue(co
ntext);
```

6.5.14 三元操作符 (If-Then-Else)

你可以在表达式中使用三元操作符来实现if-then-else的条件逻辑。下面是一个小例子：

```
String falseString = parser.parseExpression(
        "false ? 'trueExp' : 'falseExp'").getValue(String.class)
;
```

在这个例子中，因为布尔值false返回的结果一定是'falseExp'。下面是一个更实际的例子。

```
parser.parseExpression("Name").setValue(societyContext, "IEEE");
societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member o
f the ' " +
        "+ Name + ' Society' : #queryName + ' is not a member of
 the ' + Name + ' Society'";
```

6.5.15 Elvis运算符

Elvis运算符可以简化Java的三元操作符，是Groovy中使用的一种操作符。如果使用三元操作符语法你通常需要重复写两次变量名，例如：

```
String name = "Elvis Presley";
String displayName = name != null ? name : "Unknown";
```

使用Elvis运算符可以简化写法，这个符号的名字由来是它很像Elvis的发型（译者注：Elvis=Elvis Presley，猫王，著名摇滚歌手）

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue
(String.class);

System.out.println(name); // 'Unknown'
```

下面是一个复杂一点的例子:

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContex
t(tesla);

String name = parser.parseExpression("Name?:'Elvis Presley'").ge
tValue(context, String.class);

System.out.println(name); // Nikola Tesla

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(
context, String.class);

System.out.println(name); // Elvis Presley
```

6.5.16 安全引用运算符

安全引用运算符主要为了避免空指针，源于Groovy语言。很多时候你引用一个对象的方法或者属性时都需要做非空校验。为了避免此类问题、使用安全引用运算符只会返回null而不是抛出一个异常。

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

StandardEvaluationContext context = new StandardEvaluationContex
t(tesla);

String city = parser.parseExpression("PlaceOfBirth?.City").getVa
lue(context, String.class);
System.out.println(city); // Smiljan

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(con
text, String.class);

System.out.println(city); // null - does not throw NullPointerEx
ception!!!
```

> 备注：Elvis操作符可以在表达式中赋默认值，例如。在一个@Value表达式中：@Value(“#{systemProperties[‘pop3.port’] ?: 25}”)
> 上面的例子如果系统属性pop3.port已定义会直接注入，如果未定义，则返回默认值25.

6.5.17 集合筛选

该功能是SpEL中一项强大的语言特性，允许你将源集合选择其中的某几项生成另外一个集合。选择器使用语法.?[selectionExpression].通过该表达式可以过滤集合并返回原集合中的子集。例如，下面的例子我们返回inventors对象中的国籍为塞尔维亚的子集：

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
        "Members.?[Nationality == 'Serbian']").getValue(societyC
ontext);
```

筛选可以同时在list和maps上面使用。对于list来说是选择的标准是针对单个列表的每一项来比对求值，对于map来说选择的标准是针对Map的每一项（类型为Java的Map.Entry）。Map项的Key和alue都可以作为筛选的比较选项

下面的例子中表达式会返回一个新的map,包含原map中值小于27的所有子项。

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue(
);
```

除了可以返回所有被选择的元素，也可以只返回第一或者最后一项。返回第一项的选择语法是：

^[…]，返回最后一项的选择语法是 $[…].

6.5.18 集合投影

投影使得一个集合通过子表达式求值，并返回一个新的结果。投影的语法是！[projectionExpression]. 举一个通俗易懂的例子，假设我们有一个inventors 对象列表，但是我们想返回每一个inventor出生的城市列表。我们需要遍历inventor的每一项，通过 'placeOfBirth.city'来求值。下面是具体的代码例子：

```
// returns ['Smiljan', 'Idvor' ]
List placesOfBirth = (List)parser.parseExpression("Members.![pla
ceOfBirth.city]");
```

也可以在Map上使用投影、在这种场景下投影表达式会作用于Map的每一项（类型为Java的Map.Entry）。Map项的Key和alue都可以作为选择器的比较选项Map投影的结果是一个list，包含map每一项被投影表达式求值后的结果。

6.5.19 表达式模板

表达式模板运行在一段文本中混合包含一个或多个求值表达式模块。各个求值块都通过可被自定义的前后缀字符分隔，一个通用的选择是使用#{ }作为分隔符。例如：

```
String randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

求值的字符串是通过字符文本'random number is'以及#{}分隔符中的表达式求值结果拼接起来的，在这个例子中就是调用random()的结果。方法parseExpression()的第二个传入参数类型是ParserContext。ParserContext接口用来确定表达式该如何被解析、从而支持表达式的模板功能。其实现类TemplateParserContext的定义如下：

```
public class TemplateParserContext implements ParserContext {

    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }

    public boolean isTemplate() {
        return true;
    }
}
```

6.6 本章节例子中使用的类

Inventor.java

```
package org.spring.samples.spel.inventor;

import java.util.Date;
import java.util.GregorianCalendar;

public class Inventor {

    private String name;
    private String nationality;
    private String[] inventions;
    private Date birthdate;
    private PlaceOfBirth placeOfBirth;

    public Inventor(String name, String nationality) {
        GregorianCalendar c= new GregorianCalendar();
        this.name = name;
        this.nationality = nationality;
        this.birthdate = c.getTime();
    }

    public Inventor(String name, Date birthdate, String national
ity) {
        this.name = name;
        this.nationality = nationality;
        this.birthdate = birthdate;
    }

    public Inventor() {
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

    public Date getBirthdate() {
        return birthdate;
    }

    public void setBirthdate(Date birthdate) {
        this.birthdate = birthdate;
    }

    public PlaceOfBirth getPlaceOfBirth() {
        return placeOfBirth;
    }

    public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
    }

    public void setInventions(String[] inventions) {
        this.inventions = inventions;
    }

    public String[] getInventions() {
        return inventions;
    }
}
```

PlaceOfBirth.java

```
package org.spring.samples.spel.inventor;

public class PlaceOfBirth {

    private String city;
    private String country;

    public PlaceOfBirth(String city) {
        this.city=city;
    }

    public PlaceOfBirth(String city, String country) {
        this(city);
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String s) {
        this.city = s;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

}
```

Society.java

```
package org.spring.samples.spel.inventor;

import java.util.*;
```

```
public class Society {

    private String name;

    public static String Advisors = "advisors";
    public static String President = "president";

    private List&amp;lt;Inventor&amp;gt; members = new ArrayList
&amp;lt;Inventor&amp;gt;();
    private Map officers = new HashMap();

    public List getMembers() {
        return members;
    }

    public Map getOfficers() {
        return officers;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isMember(String name) {
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }

}
```

9. Spring框架下的测试

测试乃企业级软件开发的重要组成部分之一。本章专注于讲解采用 IoC 原则进行编码而给单元测试带来的好处，以及 Spring 框架对集成测试的支持如何为测试带来帮助。（对企业开发中如何进行代码测试的详尽讨论不在本文档讨论范围之内）

10. 单元测试

比起传统的Java EE开发方式，依赖注入可以弱化你的代码对容器的依赖。在基于Junit或TestNG的测试代码中，无需依赖于Spring或其他容器，你只需通过new操作符，便可以创建出组成你的应用程序的各种POJO对象。而通过mock对象（以及其它各种测试技术的综合运用），你可以将被测试的代码单独隔离开来进行测试。如果你在进行架构设计时遵循了Spring所推荐的模式，那么由此带来的诸如清晰的分层、组件化等等优点也会使你更容易地对你的代码进行单元测试。例如，通过mock DAO层或Repositry接口的实现，针对服务层对象的单元测试代码并不需要真正地访问持久层的数据。 真正的单元测试代码运行的非常快，因为并不需要一整套的运行时架构去支持测试的运转。所以在开发中强调真正的单元测试必须成为编码方法论中的一部分将会极大地提高你的生产力。 也许没有这一节内容的帮助你也能写出高效的针对IoC应用的单元测试，但是以下将要提到的 Spring 所提供的 mock 对象和测试支持类，在一些特定的场景中会对单元测试很有帮助。

10.1 Mock 对象

10.1.1 环境

`org.springframework.mock.env` 包含了对抽象 `Environment` 和 `PropertySource` 的 Mock 实现（参考 3.13.1 节 “Bean的定义文件” 和 3.13.3 节 “PropertySource抽象”）。 `MockEnvironment` 和 `MockPropertySource` 对于编写针对依赖于环境相关属性的代码的，与容器无关的测试用例很有帮助。

10.1.2 JNDI

`org.springframework.mock.jndi` 包含了JNDI SPI的实现。这一实现可以让你为你的测试套件或独立应用配置起一个简单的JNDI环境。例如，如果在测试代码和Java EE容器中JDBC的 DataSource 都绑定到同一个JNDI名字上，你就可以在测试中直接复用应用程序的代码和配置而不需做任何改动。

10.1.3 Servlet API

`org.springframework.mock.web` 包含了十分全面的Servlet API mock 对象。这些对象在测试web context，controllers 和 filters 的时候很有用。由于这些 mock 对象是有针对性地为了与 Spring 的 Web MVC 框架共同使用而编写的，因此相比起诸如 EasyMock 这种动态mock 对象或 MockObjects 这种替代性的 Servlet API mock 对象，使用起来要更为方便。

10.2 单元测试支持类

10.2.1 通用支持工具

`org.springframework.test.util` 包含了一些供单元测试和集成测试中使用的通用工具。 `ReflectionTestUtils` 是一组基于反射的方法集合。在测试包含如下一些测试用例的应用时，开发人员可以使用这些工具方法应对更改一个常量的值，设置一个非公有字段，调用一个非公有setter方法，或调用一个非公有的配置或生命周期回调方法等测试场景： 诸如JPA和Hibernate等广泛采用private或protected访问方式而非公有setter方法来访问domain entity属性的ORM（Object-Relational Mapping)框架 `@Autowired` ， `@Inject` 和 `@Resource` 等用于对 private 或 protected 字段，setter 方法和配置方法进行依赖注入的 Spring 注解。 `@PostConstruct` 和 `@PreDestroy` 等用在生命周期回调方法上的注解。 `AopTestUtils` 是一组 AOP 相关工具方法的集合。这些方法可以用于帮助获取隐藏于一重或多重 Spring 代理之后目标对象的引用。举个栗子，你使用 EasyMock 或 Mockito mock 了一个被包装在 Spring 代理之中的 bean，这时你可能需要对这个 mock 进行直接的访问，从而能够配置对此 mock 的期望行为并在稍后执行验证。关于 Spring 提供的核心 AOP 工具，请参考 AopUtils 和 AopProxyUtils 这两个类。

10.2.2 Spring MVC

`org.springframework.test.web` 包含了 `ModelAndViewAssert` 类。你可以在 Junit，TestNG或用任何测试框架编写的单元测试中使用这个类来帮助你跟 Spring MVC 框架的 `ModelAndView` 对象进行互动。

 如果想要像测试 POJO 一样来测试你的 Spring MVC 控制器，你可以将 ModelAndViewAssert 与 MockHttpServletRequest，MockHttpSession 等来自 Servlet API mocks 的 Mock 类结合使用。而假如是要把 Spring MVC 和 REST 控

制器和对 Spring MVC 的WebApplicationContext 配置结合起来进行一番彻底的集成测试，请使用 Spring MVC Test Framework 。

11. 集成测试

11.1 概述

能够在不需要部署到应用服务器或连接到其它企业基础服务的前提下做一些集成测试是很重要的。这将使你能够测试以下内容：

- Spring IoC容器上下文的正确装配。
- 使用JDBC或其它ORM工具访问数据。这将包括SQL语句、Hibernate查询和JPA实体映射的正确性等等这些内容。

Spring Framework在spring-test模块中为集成测试提供了强有力的支持。该Jar包的实际名字可能会包含发布版本号而且可能是org.springframework.test这样长的形式，这取决于你是从哪获得的（请参阅section on Dependency Management中的解释）。这个库包括了org.springframework.test包，其中包含了使用Spring容器进行集成测试的重要的类。这个测试不信赖于应用服务器和其它的发布环境。这些测试会比单元测试稍慢但比同类型的Selenium测试或信赖于发布到应用服务器的远程测试要快得多。

在Spring2.5及之后的版本，单元和集成测试支持是以注解驱动Spring TestContext框架这样的形式提供的。TestContext框架对实际使用的测试框架是不可知的的，因此可以使用包括 JUnit, TestNG等等许多测试手段。

11.2 集成测试的目标

Spring的集成测试支持有以下几点主要目标：

- 管理各个测试执行之间的 Spring IoC容器缓存
- 提供 测试配置实例的依赖注入
- 提供适合集成测试的 事务管理
- 提供辅助开发人员编写集成测试的 具备Spring特性的基础类

下面几节将解释每个目标并提供实现和配置详情的链接。

11.2.1 上下文管理和缓存

Spring TestContex框架对Spring ApplicationContext 和WebApplicationContext提供一致性加载并对它们进行缓存。对加载的上下文进行缓存提供支持是很重要的，因为启动时间是个问题——不是因为Spring自己的开销，而是被Spring容器初始化的对象需要时间去初始化。比如，一个拥有50到100个Hibernate映射文件的项目可能花费10到20秒的时间去载这些映射文件，在运行每一个测试工具中的测试用例之前都会引发开销并导致总体测试运行变缓慢，降低开发效率。

测试类通常声明一批XML的资源路径或者Groovy的配置元数据——通常在类路径中——或者是一批用于配置应用程序的注解类。这些路径或者类跟在web.xml或者其它用于生产部署的配置文件中指定的是一样的。

通常，一旦被加载过一次，ApplicationContext就将被用于每个测试中。因此启动开销在一次测试集中将只会引发一次，随后执行的测试将会快得多。在这里，“测试集”的意思是在同一个JVM的所有测试——比如说，对给定项目或者模块的一次Ant、Maven或者Gradle构建运行的所有测试。在不太可能的情况下，一个测试会破坏应用上下文并引起重新加载——比如，修改一个bean定义或者应用程序对象的状态——TestContex框架将被设置为在开始下个测试之前重新加载配置并重建应用上下文。

11.2.2 测试配置的信赖注入

当TestContext框架加载你的应用程序上下文的时候，它将通过信赖注入有选择性地配置测试实例。这为使用你的应用程序上下文中的预配置bean来建立测试配置提供了一个很方便的机制。这里有一个很大的便处就是你可以在不同的测试场景中重复使用应用程序上下文（比如，配置基于Spring管理的对象图、事务代理、数据源等等），这样省去了为每个测试用例建立复杂的测试配置的必要。

举例说明，考虑这样一个场景，我们有一个类叫做HibernateTitleRepository，它实现了Title领域实体的数据访问逻辑。我们想编写集成测试来测试以下方面：

- Spring配置：总的来说，就是与HibernateTitleRepository配置有关的一切是否正确和存在？
- Hibernate映射文件：是否所有映射都正确，并且延迟加载的设置是否准备就绪？
- HibernateTitleRepository的逻辑：此类中的配置实例是否与预期一致？

查看使用TestContext框架进行测试配置的信赖注入。

11.2.3 事务管理

测试中访问一个真实数据库的一个常见的问题是在持久层存储状态付出的努力。即使你使用开发环境的数据库，改变相应的状态也会影响将来的测试。并且，许多操作——插入或者改变持久层数据——也不能在事务之外执行（或者验证）。

TestContext框架解决了这个问题。默认行为下，这个框架将为每个测试创建并回滚一个事务。你只需简单的假定事务是存在的并写你的代码即可。如果你调用事务代理对象，他们也会根据它们配置的语义正确执行。而且，如果一个测试方法在运行相应事务时删除了选定表中的内容，事务默认情况下会进行回滚，数据库会回到测试执行前的那个状态。事务通过定义在应用程序上下文中的PlatformTransactionManager bean来得到支持。

如果你需要提交事务——通常不会这样做，但有时当你想用一个特定的测试来填充或者修改数据库时也会显得有用——TestContext框架将根据@Commit注解的指示对事务进行提交而不是回滚。

查看使用TestContext框架进行事务管理。

11.2.3 集成测试的支持类

Spring TestContext框架提供了一些支持来简化集成测试的编写。这些基础类为测试框架提供了定义良好的钩子，还有一些便利的实例变量和方法，使你能够访问：

- ApplicationContext，用于从整体上来进行显示的bean查找或者测试上下文的状态。
- JdbcTemplate，用于执行SQL语句来查询数据库。这些的查询可用于确认执行数据库相关的应用程序代码前后数据库的状态，并且Spring保证这些查询与应用程序代码在同一个事务作用域中执行。如果需要与ORM工具协同使用，请确保避免误报。

还有，你可能想用特定于你的项目的实例和方法来创建你自己自定义的，应用程序范围的超类。

查看TestContext框架的支持类。

11.3 JDBC测试支持

org.springframework.test.jdbc是包含JdbcTestUtils的包，它是一个JDBC相关的工具方法集，意在简化标准数据库测试场景。特别地，JdbcTestUtils提供以下静态工具方法：

- countRowsInTable(..)：统计给定表的行数。
- countRowsInTableWhere(..)：使用提供的where语句进行筛选统计给定表的行数。
- deleteFromTables(..)：删除特定表的全部数据。
- deleteFromTableWhere(..)：使用提供的where语句进行筛选并删除给定表的数据。
- dropTables(..)：删除指定的表。

注意 *`AbstractTransactionalJUnit4SpringContextTests`* 和 *`AbstractTransactionalTestNGSpringContextTests`* 提供了委托给前面所述的JdbcTestUtils中的方法的简便方法。

spring-jdbc模块提供了配置和启动嵌入式数据库的支持，可用于与数据库交互的集成测试中。

详见Section 15.8, “嵌入式数据库支持”和ection 15.8.5, “使用嵌入式数据库测试数据访问逻辑”。

11.4 注解

11.4.1 Spring测试注解

Spring框架提供以下Spring特定的注解集合，你可以在单元和集成测试中协同TestContext框架使用它们。请参考相应的JAVA帮助文档作进一步了解，包括默认的属性，属性别名等等。、

@BootstrapWith

@BootstrapWith是一个用于配置Spring TestContext框架如何引导的类级别的注解。具体地说，@BootstrapWith用于指定一个自定义的TestContextBootstrapper。请查看引导TestContext框架作进一步了解。

@ContextConfiguration

@ContextConfiguration定义了类级别的元数据来决定如何为集成测试来加载和配置应用程序上下文。具体地说，@ContextConfiguration声明了用于加载上下文的应用程序上下文资源路径和注解类。

资源路径通常是类路径中的XML配置文件或者Groovy脚本；而注解类通常是使用@Configuration注解的类。但是，资源路径也可以指向文件系统中的文件和脚本，解决类也可能是组件类等等。

```
@ContextConfiguration("/test-config.xml")
public class XmlApplicationContextTests {
// class body...
}
```

```
@ContextConfiguration(classes = TestConfig.class)
public class ConfigClassApplicationContextTests {
 // class body...
}
```

作为声明资源路径或注解类的替代方案或补充，@ContextConfiguration可以用于声明ApplicationContextInitializer 类。

```
@ContextConfiguration(initializers = CustomContextIntializer.cla
ss)
public class ContextInitializerTests {
 // class body...
}
```

@ContextConfiguration偶尔也被用作声明ContextLoader策略。但注意，通常你不需要显示的配置加载器，因为默认的加载器已经支持资源路径或者注解类以及初始化器。

```
@ContextConfiguration(locations = "/test-context.xml", loader =
CustomContextLoader.class)
public class CustomLoaderXmlApplicationContextTests {
 // class body...
}
```

> @ContextConfiguration默认对继承父类定义的资源路径或者配置类以及上下文初始化器提供支持。

参阅Section 11.5.4, 上下文管理和@ContextConfiguration帮助文档作进一步了解。

@WebAppConfiguration

@WebAppConfiguration是一个用于声明集成测试所加载的ApplicationContext须是WebApplicationContext的类级别的注解。测试类的@WebAppConfiguration注解只是为了保证用于测试的WebApplicationContext会被加载，它使用"file:src/main/webapp"路径默认值作为web应用的根路径（即，资源基路径）。资源基路径用于幕后创建一个 `MockServletContext`作为测试的`WebApplicationContext`的`ServletContext`。

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources")
public class WebAppTests {
 // class body...
}
```

注意@WebAppConfiguration必须和@ContextConfiguration一起使用，或者在同一个测试类，或者在测试类层次结构中。请参阅@WebAppConfiguration帮助文档作进一步了解。

@ContextHierarchy

@ContextHierarchy是一个用于为集成测试定义 `ApplicationContext`层次结构的类级别的注解。`@ContextHierarchy`应该声明一个或多个`@ContextConfiguration`实例列表，其中每一个定义上下文层次结构的一个层次。下面的例子展示了在同一个测试类中`@ContextHierarchy`的使用方法。但是，`@ContextHierarchy`一样可以用于测试类的层次结构中。

```
@ContextHierarchy({
 @ContextConfiguration("/parent-config.xml"),
 @ContextConfiguration("/child-config.xml")
})
public class ContextHierarchyTests {
 // class body...
}
```

```
@WebAppConfiguration
@ContextHierarchy({
 @ContextConfiguration(classes = AppConfig.class),
 @ContextConfiguration(classes = WebConfig.class)
})
public class WebIntegrationTests {
 // class body...
}
```

如果你想合并或者覆盖一个测试类的层次结构中的 `应用程序上下文中指定层次的配置`，你就必须在类层次中的每一个相应的层次通过为`@ContextConfiguration`的`name`属性提供与该层次相同的值的方式来显示地指定这个层次。请参阅 上下文层次关系 和 `@ContextHierarchy`帮助文档来获得更多的示例。

@ActiveProfiles

@ActiveProfiles是一个用于当集成测试加载 `ApplicationContext`的时候声明哪一个 *`bean definition profiles`* 被激活的类级别的注解。

```
@ContextConfiguration
@ActiveProfiles("dev")
public class DeveloperTests {
// class body...
}
```

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"})
public class DeveloperIntegrationTests {
// class body...
}
```

> @ActiveProfiles默认为继承激活的在超类声明的 bean definition profiles提供支持。通过实现一个自定义的 ActiveProfilesResolver并通过@ActiveProfiles的resolver属性来注册它的编程的方式来解决激活bean definition profiles问题也是可行的。
>
> 参阅使用环境profiles来配置上下文和@ActiveProfiles帮助文档作进一步了解。

参阅使用环境profiles来配置上下文和@ActiveProfiles帮助文档作进一步了解。

@TestPropertySource

@TestPropertySource是一个用于为集成测试加载ApplicationContext时配置属性文件的位置和增加到Environment中的PropertySources集中的内联属性的类级别的注解。

测试属性源比那些从系统环境或者Java系统属性以及通过@PropertySource或者编程方式声明方式增加的属性源具有更高的优先级。而且，内联属性比从资源路径加载的属性具有更高的优先级。

下面的例子展示了如何从类路径中声明属性文件。

```
@ContextConfiguration
@TestPropertySource("/test.properties")
public class MyIntegrationTests {
// class body...
}
```

面的例子展示了如何声明内联属性。

```
@ContextConfiguration
@TestPropertySource(properties = { “timezone = GMT”, “port: 4242
” })
public class MyIntegrationTests {
// class body…
}
```

@DirtiesContext

@DirtiesContext指明测试执行期间该Spring应用程序上下文已经被弄脏（也就是说通过某种方式被更改或者破坏——比如，更改单例bean的状态）。当应用程序上下文被标为”脏”，它将从测试框架缓存中被移除并关闭。因此，Spring容器将为随后需要同样配置元数据的测试而被重建。

@DirtiesContext可以在同一个类或者类层次结构中的类级别和方法级别中使用。在这个场景下，应用程序上下文将在任意此注解的方法之前或之后以及当前测试类之前或之后被标为“脏”，这取决于配置的methodMode和classMode。

下面的例子解释了在多种配置场景下什么时候上下文会被标为“脏”。

- 当在一个类中声明并将类模式设为BEFORE_CLASS，则在当前测试类之前。

```
@DirtiesContext(classMode = BEFORE_CLASS)
public class FreshContextTests {
// some tests that require a new Spring container
}
```

- 当在一个类中声明并将类模式设为AFTER_CLASS（也就是，默认的类模式），则在当前测试类之后。

```
@DirtiesContext
public class ContextDirtyingTests {
// some tests that result in the Spring container being dirtied
}
```

- 当在一个类中声明并将类模式设为BEFORE_EACH_TEST_METHOD，则在当前测试类的每个方法之前。

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)
public class FreshContextTests {
// some tests that require a new Spring container
}
```

- 当在一个类中声明并将类模式设为AFTER_EACH_TEST_METHOD，则在当前测试类的每个方法之后。

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD)
public class ContextDirtyingTests {
// some tests that result in the Spring container being dirtied
}
```

- 当在一个方法中声明并将方法模式设为BEFORE_METHOD，则在当前方法之前。

```
@DirtiesContext(methodMode = BEFORE_METHOD)
@Test
public void testProcessWhichRequiresFreshAppCtx() {
// some logic that requires a new Spring container
}
```

- 当在一个方法中声明并将方法模式设为AFTER_METHOD(也就是说，默认的方法模式），则在当前方法之后。

```
@DirtiesContext
@Test
public void testProcessWhichDirtiesAppCtx() {
// some logic that results in the Spring container being dirtied
}
```

如果@DirtiesContext被用于上下文被配置为通过@ContextHierarchy定义的上下文层次中的一部分的测试中，则hierarchyMode标志可用于控制如何声明上下文缓存。默认将使用一个穷举算法用于清除包括不仅当前层次而且与当前测试拥有共同祖先的其它上下文层次的缓存。所有在拥有共同祖先上下文的子层次的应用程序上下文都会从上下文中被移除并关闭。如果穷举算法对于特定的使用场景显得有点威力过猛，那么你可以指定一个更简单的当前层算法来代替，如下所。

```
@ContextHierarchy({
@ContextConfiguration("/parent-config.xml"),
@ContextConfiguration("/child-config.xml")
})
public class BaseTests {
// class body...
}

public class ExtendedTests extends BaseTests {

@Test
@DirtiesContext(hierarchyMode = CURRENT_LEVEL)
public void test() {
// some logic that results in the child context being dirtied
}
}
```

参阅DirtiesContext.HierarchyMode帮助文档以获得穷举和当前层算法更详细的了解。

@TestExecutionListeners

@TestExecutionListeners定义了一个类级别的元数据，用于配置需要用TestContextManager进行注册的 `TestExecutionListener`实现。通常，`@TestExecutionListeners`与`@ContextConfiguration`一起使用。

```
@ContextConfiguration
@TestExecutionListeners({CustomTestExecutionListener.class, Anot
herTestExecutionListener.class})
public class CustomTestExecutionListenerTests {
// class body...
}
```

@TestExecutionListeners默认支持继承监听器。参阅帮助文档获得示例和更详细的了解。

@Commit

@Commit指定事务性的测试方法在测试方法执行完成后对事务进行提交。@Commit可以用作@Rollback(false)的直接替代，以更好的传达代码的意图。和@Rollback一样，@Commit可以在类层次或者方法层级声明。

```
@Commit
@Test
public void testProcessWithoutRollback() {
// ...
}
```

@Rollback

@Rollback指明当测试方法执行完毕的时候是否对事务性方法中的事务进行回滚。如果为true,则进行回滚；否则，则提交（请参加@Commit）。在Spring TestContext框架中，集成测试默认的Rollback语义为true，即使你不显示的指定它。

当被声明为方法级别的注解，则@Rollback为特定的方法指定回滚语义，并覆盖类级别的@Rollback和@Commit语义。

```
@Rollback(false)
@Test
public void testProcessWithoutRollback() {
// …
}
```

@BeforeTransaction

@BeforeTransaction指明通过Spring的@Transactional注解配置为需要在事务中执行的测试方法在事务开始之前先执行注解的void方法。从Spring框架4.3版本起，@BeforeTransaction方法不再需要为public并可能被声明为基于Java8的接口的默认方法。

```
@BeforeTransaction
void beforeTransaction() {
// logic to be executed before a transaction is started
}
```

@AfterTransaction

@AfterTransaction指明通过Spring的@Transactional注解配置为需要在事务中执行的测试方法在事务结束之后执行注解的void方法。从Spring框架4.3版本起，@AfterTransaction方法不再需要为public并可能被声明为基于Java8的接口的默认方法。

```
@AfterTransaction
void afterTransaction() {
// logic to be executed after a transaction has ended
}
```

@Sql

@Sql用于注解测试类或者测试方法，以让在集成测试过程中配置的SQL脚本能够在给定的的数据库中得到执行。

```
@Test
@Sql({"/test-schema.sql", "/test-user-data.sql"})
public void userTest {
// execute code that relies on the test schema and test data
}
```

请参阅通过@sql声明执行的SQL脚本作进一步了解。

@SqlConfig

@SqlConfig定义了用于决定如何解析和执行通过@Sql注解配置的SQL脚本。

```
@Test
@Sql(
scripts = "/test-user-data.sql",
config = @SqlConfig(commentPrefix = "`", separator = "@@")
)
public void userTest {
// execute code that relies on the test data
}
```

@SqlGroup

@SqlGroup是一个用于聚合几个@Sql注解的容器注解。@SqlGroup可以直接使用，通过声明几个嵌套的@Sql注解，也可以与Java8的可重复注解支持协同使用，即简单地在同一个类或方法上声明几个@Sql注解，隐式地产生这个容器注解。

```
@Test
@SqlGroup({
 @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentP
refix = "`")),
 @Sql("/test-user-data.sql")
)}
public void userTest {
 // execute code that uses the test schema and test data
}
```

11.4.2 标准注解支持

以下注解为Spring TestContext 框架所有的配置提供标准语义支持。注意这些注解不仅限于测试，可以用在Spring框架的任意地方。

- @Autowired
- @Qualifier
- @Resource(javax.annotation)如果*JSR-250*存在
- @ManagedBean(javax.annotation)如果*JSR-250*存在
- @Inject(javax.inject)如果*JSR-330*存在
- @Named(javax.inject)如果*JSR-330*存在
- @PersistenceContext(javax.persistence)如果JPA存在
- @PersistenceUnit(javax.persistence)如果JPA存在
- @Required
- @Transactional

> 在Spring TestContext 框架中，@PostConstruct 和 @PreDestroy 可以通过标准语义在配置于应用程序上下文的任意应用程序组件中使用; 但是, 这些生命周期注解在实际测试类中只有很有限的作用。如果一个测试类的方法被注解为@PostConstruct，这个方法将在test框架中的任何before方法（也就是被JUnit中的@Before注解方法）调用之前被执行, 这个规则将被应用于测试类的每个方法。另一方面，如果一个测试类的方法被注解为 @PreDestroy，这个方法将永远不会被执行。因为建议在测试类中使用test 框架的测试生命周期回调来代替使用@PostConstruct and @PreDestroy。

11.4.3 Spring JUnit 4 测试注解

@IfProfileValue指明该测试只在特定的测试环境中被启用。如果ProfileValueSource配置的name属性与此注解配置的name属性一致，这该测试将被启用。否则，该测试将被禁用并忽略。

@IfProfileValue可以用在类级别、方法级别或者两个同时。使用类级别的@IfProfileValue注解优先于当前类或其子类的任意方法的使用方法级别的注解。有@IfProfileValue注解意味着则测试被隐式开启。这与JUnit4的@Ignore注解是相类似的，除了使用@Ignore注解是用于禁用测试的之外。

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation")
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
// some logic that should run only on Java VMs from Oracle Corpo
ration
}
```

或者，你可以 配置@IfProfileValue使用 `values列表（或语义）来实现JUnit 4环境中的类似TestNG对测试组的支持。`

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integ
ration-tests"})
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups()
 {
// some logic that should run only for unit and integration test
 groups
}
```

@ProfileValueSourceConfiguration

@ProfileValueSourceConfiguration是类级别注解，用于当获取通过@IfProfileValue配置的profile值时指定使用什么样的ProfileValueSource类型。如果一个测试没有指定@ProfileValueSourceConfiguration，那么默认使用SystemProfileValueSource。

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class)
public class CustomProfileValueSourceTests {
// class body...
}
```

@Timed

@Timed用于指明被注解的测试必须在指定的时限（毫秒）内结束。如果测试超过指定时限，就当作测试失败。

时限包括测试方法本身所耗费的时间，包括任何重复（请查看@Repeat）及任意初始化和销毁所用的时间。

```
@Timed(millis=1000)
public void testProcessWithOneSecondTimeout() {
 // some logic that should not take longer than 1 second to exec
ute
}
```

Spring的@Timed注解与JUnit 4的@Test(timeout=...)支持相比具有不同的语义。确切地说，由于在JUnit 4中处理方法执行超时的方式（也就是，在独立纯程中执行该测试方法），如果一个测试方法执行时间太长，@Test(timeout=...)将直接判定该测

试失败。而Spring的@Timed则不直接判定失败而是等待测试完成。

@Repeat

@Repeat指明该测试方法需被重复执行。注解指定该测试方法被重复的次数。重复的范围包括该测试方法自身也包括相应的初始化和销毁方法。

```
@Repeat(10)
@Test
public void testProcessRepeatedly() {
 // ...
}
```

11.4.4 Meta-Annotation Support for Testing

可以将大部分测试相关的注解当作meta-annotations使用，以创建自定义组合注解来减少测试集中的重复配置。

下面的每个都可以在TestContext框架中被当作meta-annotations使用。

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`
- `@SqlGroup`
- `@Repeat`

- `@Timed`
- `@IfProfileValue`
- `@ProfileValueSourceConfiguration`

例如，如果发现我们在基于JUnit 4的测试集中重复以下配置...

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-con
fig.xml"})
@ActiveProfiles("dev")
@Transactional
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-con
fig.xml"})
@ActiveProfiles("dev")
@Transactional
public class UserRepositoryTests { }
```

我们可以通过一个自定义的组合注解来减少上述的重复量，将通用的测试配置集中起来，就像这样：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"/app-config.xml", "/test-data-access-con
fig.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTest { }
```

然后我们就可以像下面一样使用我们自定义的@TransactionalDevTest注解来简化每个类的配置：

```
@RunWith(SpringRunner.class)
@TransactionalDevTest
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@TransactionalDevTest
public class UserRepositoryTests { }
```

想获得详情，请查看Spring注解编程模型

14. DAO支持

14.1 介绍

在Spring中数据访问对象(DAO)旨在使JDBC,Hibernate，JPA或JDO等数据访问技术有一致的处理方法，并且方法尽可能简单。
这样就可以很容易地切换上述持久化技术，并且切换过程无需担心每种技术的特有异常。

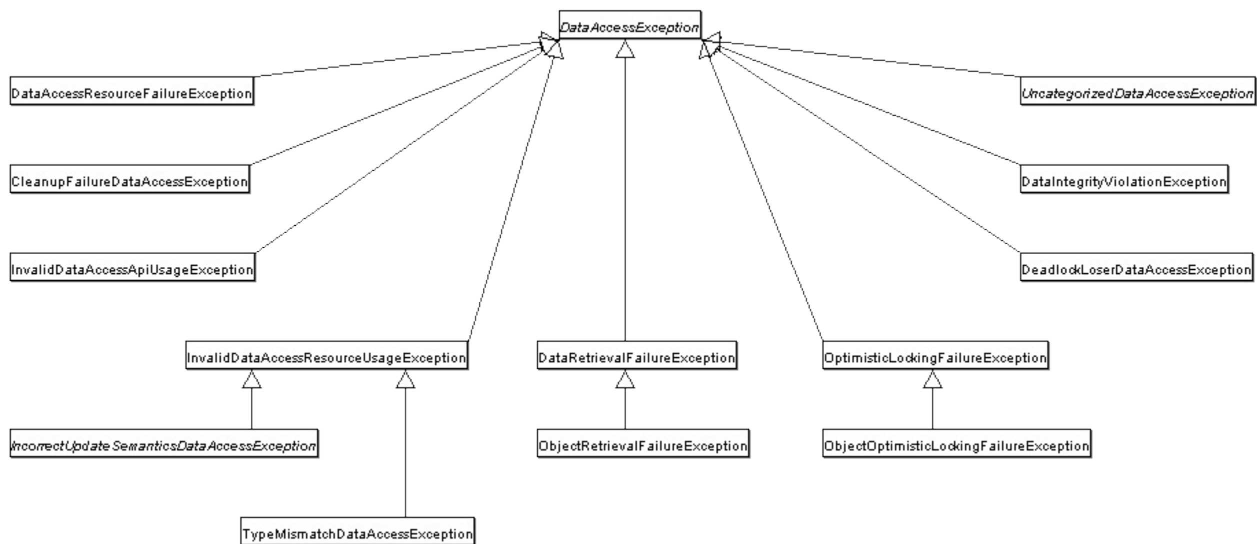
14.2 一致的异常层级

Spring为技术特定异常提供了一个适当的转化，例如：SQLException 所属异常类层级用DataAccessException 作为根异常，这些异常包括了原来的异常，所以不会有丢失可能出错信息的风险。

除了JDBC异常，Spring也包含了Hibernate特定异常，将它们转换为一组集中的运行时异常(对JDO 和 JPA 异常也是如此)，在合适的层次上处理多数不可恢复的持久化异常，而不会在dao上产生繁琐的catch-throw块和异常声明（仍然可以在认为合适的地方捕获和处理异常）。向上面提到的一样，JDBC异常（包括数据方言）也都转化为相同的层级结构，意味着在一个统一的项目模型中你也可以执行一些JDBC操作。

以上列举的Spring的各种模板类支持各种ORM框架。如果使用基于拦截器的类，那么我们的程序必须关心并处理HibernateExceptions和JDOExceptions本身，最好是通过分别授权给SessionFactoryUtils' `convertHibernateAccessException(..)或convertJdoAccessException()方法。这些方法将这些异常转化为与org.springframework.dao中异常层级兼容的异常。由于JDOExceptions 没有被检查，它可以被简单的抛出，这也牺牲了DAO在异常上的抽象。

下图展示了Spring提供的异常层级，（请注意：在这张图上显示出来的类层级仅仅是整个DataAccessException 的一个子集）



14.3 配置 DAO 或 Repository 类的注解

使用@Repository注解是数据访问对象（DAOs）或库能提供异常转换的最好方式，这个注解还允许组件扫描，查找并配置你的 DAOs 和库，并且不需要为它们提供 XML 配置文件。

```
@Repository
public class SomeMovieFinder implements MovieFinder {
// ...
}
```

任何DAO或库实现都需要访问持久的源，依赖于持久化技术的使用；例如：一个基于JDBC的库需要访问一个JDBC DataSource，一个基于JPA的库需要访问一个EntityManager，最简单的方式就是使用 @Autowired, @Inject, @Resource 或 @PersistenceContext 这些注解中的一个完成资源的依赖注入，这是一个JPA库的例子：

```
@Repository
public class JpaMovieFinder implements MovieFinder {

@PersistenceContext
private EntityManager entityManager;

// ...

}
```

如果你使用传统的Hibernate API，你可以注入SessionFactory：

```
@Repository
public class HibernateMovieFinder implements MovieFinder {

private SessionFactory sessionFactory;

@Autowired
public void setSessionFactory(SessionFactory sessionFactory) {
this.sessionFactory = sessionFactory;
}

// ...

}
```

最后一个例子我们将在这里展示典型的JDBC支持，你将会在初始化方法中注入DataSource ，在初始化方法中，你将使用这个DataSource创建一个JdbcTemplate和其他与SimpleJdbcCall相似的数据访问支持类。

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

private JdbcTemplate jdbcTemplate;

@Autowired
public void init(DataSource dataSource) {
this.jdbcTemplate = new JdbcTemplate(dataSource);
}

// ...

}
```

15.使用JDBC实现数据访问

15.1 介绍Spring JDBC框架

表格13.1很清楚的列举了Spring框架针对JDBC操作做的一些抽象和封装。里面区分了哪些操作Spring已经帮你做好了、哪些操作是应用开发者需要自己负责的.

表13.1. Spring JDBC – 框架和应用开发者各自分工

操作	Spring	开发者
定义连接参数		X
打开连接	X	
指定SQL语句		X
声明参数和提供参数值		X
准备和执行语句	X	
返回结果的迭代（如果有）	X	
具体操作每个迭代		X
异常处理	X	
事务处理	X	
关闭连接、语句和结果集	X	

一句话、Spring帮你屏蔽了很多JDBC底层繁琐的API操作、让你更方便的开发

15.1.1 选择一种JDBC数据库访问方法

JDBC数据库访问有几种基本的途径可供选择。除了JdbcTemplate的三种使用方式外，新的SimpleJdbcInsert和SimplejdbcCall调用类通过优化数据库元数据（来简化JDBC操作），还有一种更偏向于面向对象的RDBMS对象风格的方法、有点类似于JDO的查询设计。即使你已经选择了其中一种方法、你仍然可以混合使用另外一种方法的某一个特性。所有的方法都需要JDBC2.0兼容驱动的支持，一些更高级的特性则需要使用JDBC3.0驱动支持。

JdbcTemplate 是经典的Spring JDBC访问方式，也是最常用的。这是“最基础”的方式、其他所有方式都是在 JdbcTemplate的基础之上封装的。

NamedParameterJdbcTemplate 在原有JdbcTemplate的基础上做了一层包装支持命名参数特性、用于替代传统的JDBC“？”占位符。当SQL语句中包含多个参数时使用这种方式能有更好的可读性和易用性

SimpleJdbcInsert和SimpleJdbcCall操作类主要利用JDBC驱动所提供的数据库元数据的一些特性来简化数据库操作配置。这种方式简化了编码、你只需要提供表或者存储过程的名字、以及和列名相匹配的参数Map。但前提是数据库需要提供足够的元数据。如果数据库没有提供这些元数据，需要开发者显式配置参数的映射关系。

RDBMS对象的方式包含MappingSqlQuery, SqlUpdate和StoredProcedure，需要你在初始化应用数据访问层时创建可重用和线程安全的对象。这种方式设计上类似于JDO查询、你可以定义查询字符串，声明参数及编译查询语句。一旦完成这些工作之后，执行方法可以根据不同的传入参数被多次调用。

15.1.2 包层级

Spring的JDBC框架一共包含4种不同类型的包、包括core,datasource,object和support.

org.springframework.jdbc.core包含JdbcTemplate 类和它各种回调接口、外加一些相关的类。它的一个子包org.springframework.jdbc.core.simple包含SimpleJdbcInsert和SimpleJdbcCall等类。另一个叫org.springframework.jdbc.core.namedparam的子包包含NamedParameterJdbcTemplate及它的一些工具类。详见：
15.2：“使用JDBC核心类控制基础的JDBC处理过程和异常处理机制”
15.4：“JDBC批量操作”
15.5：“利用SimpleJdbc 类简化JDBC操作”.

org.springframework.jdbc.datasource包包含DataSource数据源访问的工具类，以及一些简单的DataSource实现用于测试和脱离JavaEE容器运行的JDBC代码。子包org.springfamework.jdbc.datasource.embedded提供Java内置数据库例如HSQL, H2, 和Derby的支持。详见：
15.3：“控制数据库连接”
15.8：“内置数据库支持”.

org.springframework.jdbc.object包含用于在RDBMS查询、更新和存储过程中创建线程安全及可重用的对象类。详见15.6：“像Java对象那样操作JDBC”；这种方式类似于JDO的查询方式，不过查询返回的对象是与数据库脱离的。此包针对JDBC

做了很多上层封装、而底层依赖于org.springframework.jdbc.core包。

org.springframework.jdbc.support包含SQLException的转换类和一些工具类。JDBC处理过程中抛出的异常会被转换成org.springframework.dao里面定义的异常类。这意味着SpringJDBC抽象层的代码不需要实现JDBC或者RDBMS特定的错误处理方式。所有转换的异常都没有被捕获，而是让开发者自己处理异常、具体的话既可以捕获异常也可以直接抛给上层调用者
详见：15.2.3：“SQL异常转换器”.

15.2 使用JDBC核心类控制基础的JDBC处理过程和异常处理机制

15.2.1 JdbcTemplate

JdbcTemplate是JDBC core包里面的核心类。它封装了对资源的创建和释放，可以帮你避免忘记关闭连接等常见错误。它也包含了核心JDBC工作流的一些基础工作、例如执行和声明语句，而把SQL语句的生成以及查询结果的提取工作留给应用代码。JdbcTemplate执行查询、更新SQL语句和调用存储过程，运行结果集迭代和抽取返回参数值。它也可以捕获JDBC异常并把它们转换成更加通用、解释性更强的异常层次结构、这些异常都定义在org.springframework.dao包里面。

当你在代码中使用了JdbcTemplate类，你只需要实现回调接口。PreparedStatementCreator回调接口通过传入的Connection类（该类包含SQL和任何必要的参数）创建已声明的语句。CallableStatementCreator也提供类似的方式、该接口用于创建回调语句。RowCallbackHandler用于获取结果集每一行的值。

可以在DAO实现类中通过传入DataSource引用来完成JdbcTemplate的初始化；也可以在Spring IOC容器里面配置、作为DAO bean的依赖Bean配置。

> 备注：DataSource最好在Spring IOC容器里作为Bean配置起来。在上面第一种情况下，DataSource bean直接传给相关的服务；第二种情况下DataSource bean传递给JdbcTemplate bean。

JdbcTemplate中使用的所有SQL以“DEBUG”级别记入日志（一般情况下日志的归类是JdbcTemplate对应的全限定类名，不过如果需要对JdbcTemplate进行定制的话，可能是它的子类名）

JdbcTemplate 使用示例

这一节提供了JdbcTemplate类的一些使用例子。这些例子没有囊括JdbcTemplate可提供的所有功能；全部功能和用法请详见相关的javadocs.

查询 **(SELECT)**

下面是一个简单的例子、用于获取关系表里面的行数

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*)
 from t_actor", Integer.class);
```

使用绑定变量的简单查询：

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Int
eger.class, "Joe");
```

String查询：

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
```

查询和填充领域模型：

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?"
,
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws
 SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

查询和填充多个领域对象：

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws
 SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

如果上面的两段代码实际存在于相同的应用中，建议把RowMapper匿名类中重复的代码抽取到单独的类中（通常是一个静态类），方便被DAO方法引用。例如，上面的代码例子更好的写法如下：

```
public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_nam
e from t_actor", new ActorMapper());
}

private static final class ActorMapper implements RowMapper<Acto
r> {

    public Actor mapRow(ResultSet rs, int rowNum) throws SQLExce
ption {
        Actor actor = new Actor();
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

使用**jdbcTemplate**实现增删改

你可以使用update(..)方法实现插入，更新和删除操作。参数值可以通过可变参数或者封装在对象内传入。

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?,
?)",
        "Leonor", "Watling");
```

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

```
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

其他**jdbcTemplate**操作

你可以使用execute(..)方法执行任何SQL，甚至是DDL语句。这个方法可以传入回调接口、绑定可变参数数组等。

```
this.jdbcTemplate.execute("create table mytable (id integer, nam
e varchar(100))");
```

下面的例子调用一段简单的存储过程。更复杂的存储过程支持文档后面会有描述。

```
this.jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        Long.valueOf(unionId));
```

JdbcTemplate 最佳实践

JdbcTemplate实例一旦配置之后是线程安全的。这点很重要因为这样你就能够配置JdbcTemplate的单例，然后安全的将其注入到多个DAO中（或者repositories）。JdbcTemplate是有状态的，内部存在对DataSource的引用，但是这种状态不是会话状态。

使用JdbcTemplate类的常用做法是在你的Spring配置文件里配置好一个DataSource，然后将其依赖注入进你的DAO类中（NamedParameterJdbcTemplate也是如此）。JdbcTemplate在DataSource的Setter方法中被创建。就像如下DAO类的写法一样：

```
public class JdbcCorporateEventDao implements CorporateEventDao
{

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the Corpora
teEventDao follow...
}
```

相关的配置是这样的：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context
"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-con
text.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporat
eEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDa
taSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverCla
ssName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

另一种替代显式配置的方式是使用component-scanning和注解注入。在这个场景下需要添加@Repository注解（添加这个注解可以被component-scanning扫描到），同时在DataSource的Setter方法上添加@Autowired注解：

```
@Repository
public class JdbcCorporateEventDao implements CorporateEventDao
{

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the Corpora
teEventDao follow...
}
```

相关的XML配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context
"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-con
text.xsd">

    <!-- Scans within the base package of the application for @C
omponent classes to configure as beans -->
    <context:component-scan base-package="org.springframework.do
cs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDa
taSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverCla
ssName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

如果你使用Spring的JdbcDaoSupport类，许多JDBC相关的DAO类都从该类继承过来，这个时候相关子类需要继承JdbcDaoSupport类的setDataSource方法。当然你也可以选择不从这个类继承，JdbcDaoSupport本身只是提供一些便利性。

无论你选择上面提到的哪种初始方式，当你在执行SQL语句时一般都不需要重新创建JdbcTemplate 实例。JdbcTemplate一旦被配置后其实例都是线程安全的。当你的应用需要访问多个数据库时你可能也需要多个JdbcTemplate实例，相应的也需要多个DataSources，同时对应多个JdbcTemplates配置。

15.2.2 NamedParameterJdbcTemplate

NamedParameterJdbcTemplate 提供对JDBC语句命名参数的支持，而普通的JDBC语句只能使用经典的 '?' 参数。NamedParameterJdbcTemplate内部包装了JdbcTemplate，很多功能是直接通过JdbcTemplate来实现的。本节主要描述NamedParameterJdbcTemplate不同于JdbcTemplate 的点；即通过使用命名参数来操作JDBC

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemp
late(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name
= :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSour
ce("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, n
amedParameters, Integer.class);
}
```

上面代码块可以看到SQL变量中命名参数的标记用法，以及namedParameters变量的相关赋值（类型为MapSqlParameterSource）

除此以外，你还可以在NamedParameterJdbcTemplate中传入Map风格的命名参数及相关的值。NamedParameterJdbcTemplate类从NamedParameterJdbcOperations接口实现的其他方法用法是类似的，这里就不一一叙述了。

下面是一个Map风格的例子：

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemp
late(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name
= :first_name";

    Map<String, String> namedParameters = Collections.singletonM
ap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, n
amedParameters,  Integer.class);
}
```

与NamedParameterJdbcTemplate相关联的SqlParameterSource接口提供了很有用的功能（两者在同一个包里面）。在上面的代码片段中你已经看到了这个接口的一个实现例子（就是MapSqlParameterSource类）。SqlParameterSource类是NamedParameterJdbcTemplate
类的数值值来源。MapSqlParameterSource实现非常简单、只是适配了java.util.Map，其中Key就是参数名字，Value就是参数值。

另外一个SqlParameterSource 的实现是BeanPropertySqlParameterSource类。这个类封装了任意一个JavaBean（也就是任意符合JavaBen规范的实例），在这个实现中，使用了JavaBean的属性作为命名参数的来源。

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemp
late(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of t
he above 'Actor' class
    String sql = "select count(*) from T_ACTOR where first_name
= :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlPara
meterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForObject(sql, n
amedParameters, Integer.class);
}
```

之前提到过NamedParameterJdbcTemplate本身包装了经典的JdbcTemplate模板。如果你想调用只存在于JdbcTemplate类中的方法，你可以使用getJdbcOperations（）方法、该方法返回JdbcOperations接口，通过这个接口你可以调用内部JdbcTemplate的方法。

NamedParameterJdbcTemplate 类在应用上下文的使用方式也可见：“JdbcTemplate最佳实践”

15.2.3 SQLExceptionTranslator

SQLExceptionTranslator接口用于在SQLExceptions和spring自己的org.springframework.dao.DataAccessException之间做转换，要处理批量更新或者从文件中这是为了屏蔽底层的数据访问策略。其实现可以是比较通用的（例如，使用JDBC的SQLState编码），或者是更精确专有的（例如，使用Oracle的错误类型编码）

SQLExceptionTranslator 接口的默认实现是 SQLErrorCodeSQLExceptionTranslator，该实现使用的是指定数据库厂商的错误编码，因为要比SQLState的实现更加精确。错误码转换过程基于JavaBean类型的SQLErrorCodes。这个类通过SQLErrorCodesFactory创建和返回，SQLErrorCodesFactory是一个基于sql-error-codes.xml配置内容来创建SQLErrorCodes的工厂类。该配置中的数据库厂商代码基于Database MetaData信息中返回的数据库产品名（DatabaseProductName），最终使用的就是你正在使用的实际数据库中错误码。

SQLErrorCodeSQLExceptionTranslator按以下的顺序来匹配规则：

> 备注：SQLErrorCodesFactory是用于定义错误码和自定义异常转换的缺省工厂类。错误码参照Classpath下配置的sql-error-codes.xml文件内容，相匹配的SQLErrorCodes实例基于正在使用的底层数据库的元数据名称

- 是否存在自定义转换的子类。通常直接使用SQLErrorCodeSQLExceptionTranslator就可以了，因此此规则一般不会生效。只有你真正自己实现了一个子类才会生效。
- 是否存在SQLExceptionTranslator接口的自定义实现，通过SQLErrorCodes类的customSqlExceptionTranslator属性指定
- SQLErrorCodes的customTranslations属性数组、类型为CustomSQLErrorCodesTranslation类实例列表、能否被匹配到
- 错误码被匹配到
- 使用兜底的转换器。SQLExceptionSubclassTranslator是缺省的兜底转换器。如果此转换器也不存在的话只能使用SQLStateSQLExceptionTranslator

你可以继承SQLErrorCodeSQLExceptionTranslator：

```
public class CustomSQLErrorCodesTranslator extends SQLErrorCodeS
QLExceptionTranslator {

    protected DataAccessException customTranslate(String task, S
tring sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sq
lex);
        }
        return null;
    }
}
```

这个例子中，特定的错误码-12345被识别后单独转换，而其他的错误码则通过默认的转换器实现来处理。在使用自定义转换器时，有必要通过setExceptionTranslator方法传入JdbcTemplate ，并且使用JdbcTemplate来做所有的数据访问处理。下面是一个如何使用自定义转换器的例子

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

    // create a JdbcTemplate and set data source
    this.jdbcTemplate = new JdbcTemplate();
    this.jdbcTemplate.setDataSource(dataSource);

    // create a custom translator and set the DataSource for the
 default translation lookup
    CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTr
anslator();
    tr.setDataSource(dataSource);
    this.jdbcTemplate.setExceptionTranslator(tr);

}

public void updateShippingCharge(long orderId, long pct) {
    // use the prepared JdbcTemplate for this update
    this.jdbcTemplate.update("update orders" +
        " set shipping_charge = shipping_charge * ? / 100" +
        " where id = ?", pct, orderId);
}
```

自定义转换器需要传入dataSource对象为了能够获取sql-error-codes.xml定义的错误码

15.2.4 执行SQL语句

执行一条SQL语句非常方便。你只需要依赖DataSource和JdbcTemplate，包括JdbcTemplate提供的工具方法。
下面的例子展示了如何创建一个新的数据表，虽然只有几行代码、但已经完全可用了：

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void doExecute() {
        this.jdbcTemplate.execute("create table mytable (id inte
ger, name varchar(100))");
    }
}
```

15.2.5 查询

一些查询方法会返回一个单一的结果。使用queryForObject(..)返回结果计数或特定值。当返回特定值类型时，将Java类型作为方法参数传入、最终返回的JDBC类型会被转换成相应的Java类型。如果这个过程中间出现类型转换错误，则会抛出InvalidDataAccessApiUsageException的异常。下面的例子包含两个查询方法，一个返回int类型、另一个返回了String类型。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCount() {
        return this.jdbcTemplate.queryForObject("select count(*)
 from mytable", Integer.class);
    }

    public String getName() {
        return this.jdbcTemplate.queryForObject("select name fro
m mytable", String.class);
    }
}
```

除了返回单一查询结果的方法外，其他方法返回一个列表、列表中每一项代表查询返回的行记录。其中最通用的方式是queryForList(..)，返回一个列表，列表每一项是一个Map类型，包含数据库对应行每一列的具体值。下面的代码块给上面的例子添加一个返回所有行的方法：

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public List<Map<String, Object>> getList() {
    return this.jdbcTemplate.queryForList("select * from mytable
");
}
```

返回的列表结果数据格式是这样的：

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

15.2.6 更新数据库

下面的例子根据主键更新其中一列值。在这个例子中，一条SQL语句包含行参数的占位符。参数值可以通过可变参数或者对象数组传入。元数据类型需要显式或者自动装箱成对应的包装类型

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void setName(int id, String name) {
        this.jdbcTemplate.update("update mytable set name = ? wh
ere id = ?", name, id);
    }
}
```

15.2.7 获取自增Key

update()方法支持获取数据库自增Key。这个支持已成为JDBC3.0标准之一、更多细节详见13.6章。这个方法使用PreparedStatementCreator作为其第一个入参，该类可以指定所需的insert语句。另外一个参数是KeyHolder，包含了更新操作成功之后产生的自增Key。这不是标准的创建PreparedStatement 的方式。下面的例子可以在Oracle上面运行，但在其他平台上可能就不行了。

```
final String INSERT_SQL = "insert into my_test (name) values(?)"
;
final String name = "Rob";

KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(
    new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connect
ion connection) throws SQLException {
            PreparedStatement ps = connection.prepareStatement(I
NSERT_SQL, new String[] {"id"});
            ps.setString(1, name);
            return ps;
        }
    },
    keyHolder);

// keyHolder.getKey() now contains the generated key
```

15.3 控制数据库连接

15.3.1 DataSource

Spring用DataSource来保持与数据库的连接。DataSource是JDBC规范的一部分同时是一种通用的连接工厂。它使得框架或者容器对应用代码屏蔽连接池或者事务管理等底层逻辑。作为开发者，你无需知道连接数据库的底层逻辑；这只是创建datasource的管理员该负责的模块。在开发测试过程中你可能需要同时扮演双重角色，但最终上线时你不需要知道生产数据源是如何配置的。

当使用Spring JDBC时，你可以通过JNDI获取数据库数据源、也可以利用第三方依赖包的连接池实现来配置。比较受欢迎的三方库有Apache Jakarta Commons DBCP 和 C3P0。在Spring产品内，有自己的数据源连接实现，但仅仅用于测试目的，同时并没有使用到连接池。

这一节使用了Spring的DriverManagerDataSource实现、其他更多的实现会在后面提到。

> 注意：仅仅使用DriverManagerDataSource类只是为了测试目的、因为此类没有连接池功能，因此在并发连接请求时性能会比较差

通过DriverManagerDataSource获取数据库连接的方式和传统JDBC是类似的。首先指定JDBC驱动的类全名，DriverManager 会据此来加载驱动类。接下来、提供JDBC驱动对应的URL名称。（可以从相应驱动的文档里找到具体的名称）。然后传入用户名和密码来连接数据库。下面是一个具体配置DriverManagerDataSource连接的Java代码块:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource
();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

接下来是相关的XML配置：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource
.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassNa
me}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

下面的例子展示的是DBCP和C3P0的基础连接配置。如果需要连接更多的连接池选项、请查看各自连接池实现的具体产品文档

DBCP配置：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSo
urce" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassNa
me}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

C3P0配置：

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledData
Source" destroy-method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"
/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

15.3.2 DataSourceUtils

DataSourceUtils类是一个方便有用的工具类，提供了从JNDI获取和关闭连接等有用的静态方法。它支持线程绑定的连接、例如：使用DataSourceTransactionManager的时候，将把数据库连接绑定到当前的线程上。

15.3.3 SmartDataSource

实现SmartDataSource接口的实现类需要能够提供到关系数据库的连接。它继承了DataSource接口，允许使用它的类查询是否在某个特定的操作后需要关闭连接。这在当你需要重用连接时比较有用。

15.3.4 AbstractDataSource

AbstractDataSource是Spring DataSource实现的基础抽象类，封装了DataSource的基础通用功能。你可以继承AbstractDataSource自定义DataSource 实现。

15.3.5 SingleConnectionDataSource

SingleConnectionDataSource实现了SmartDataSource接口、内部封装了一个在每次使用后都不会关闭的单一连接。显然，这种场景下无法支持多线程。

为了防止客户端代码误以为数据库连接来自连接池（就像使用持久化工具时一样）错误的调用close方法，你应将suppressClose设置为true。这样，通过该类获取的将是代理连接（禁止关闭）而不是原有的物理连接。需要注意你不能将这个类强制转换成Oracle等数据库的原生连接。

这个类主要用于测试目的。例如，他使得测试代码能够脱离应用服务器，很方便的在单一的JNDI环境下调试。和DriverManagerDataSource相反，它总是重用相同的连接，这是为了避免在测试过程中创建过多的物理连接。

15.3.6 DriverManagerDataSource

DriverManagerDataSource类实现了标准的DataSource接口，可以通过Java Bean属性来配置原生的JDBC驱动，并且每次都返回一个新的连接。

这个实现对于测试和JavaEE容器以外的独立环境比较有用，无论是作为一个在Spring IOC容器内的DataSource Bean，或是在单一的JNDI环境中。由于Connection.close()仅仅只是简单的关闭数据库连接，因此任何能够操作DataSource的持久层代码都能很好的工作。但是，使用JavaBean类型的连接池，比如commons-dbcp往往更简单、即使是在测试环境下也是如此，因此更推荐commons-dbcp。

15.3.7 TransactionAwareDataSourceProxy

TransactionAwareDataSourceProxy会创建一个目标DataSource的代理，内部包装了DataSource，在此基础上添加了Spring事务管理功能。有点类似于JavaEE服务器中提供的JNDI事务数据源。

> 注意：一般情况下很少用到这个类，除非现有代码在被调用的时候需要一个标准的 JDBC DataSource接口实现作为参数。在这种场景下，使用proxy可以仍旧重用老代码，同时能够有Spring管理事务的能力。更多的场景下更推荐使用JdbcTemplate和DataSourceUtils等更高抽象的资源管理类.

更多细节请查看TransactionAwareDataSourceProxy的JavaDoc）

15.3.8 DataSourceTransactionManager

DataSourceTransactionManager类实现了PlatformTransactionManager接口。它将JDBC连接从指定的数据源绑定到当前执行的线程中，
允许一个线程连接对应一个数据源。

应用代码需要通过DataSourceUtils.getConnection(DataSource) 来获取JDBC连接，而不是通过JavaEE标准的DataSource.getConnection来获取。它会抛出org.springframework.dao的运行时异常而不是编译时SQL异常。所有框架类像JdbcTemplate都默认使用这个策略。如果不需要和这个DataSourceTransactionManager类一起使用，DataSourceUtils 提供的功能跟一般的数据库连接策略没有什么两样，因此它可以在任何场景下使用。

DataSourceTransactionManager支持自定义隔离级别，以及JDBC查询超时机制。为了支持后者，应用代码必须在每个创建的语句中使用JdbcTemplate或是调用DataSourceUtils.applyTransactionTimeout(..)方法

在单一的资源使用场景下它可以替代JtaTransactionManager，不需要要求容器去支持JTA。如果你严格遵循连接查找的模式的话、可以通过配置来做彼此切换。JTA本身不支持自定义隔离级别！

15.4 JDBC批量操作

大多数JDBC驱动在针对同一SQL语句做批处理时能够获得更好的性能。批量更新操作可以节省数据库的来回传输次数。

15.4.1 使用JdbcTemplate来进行基础的批量操作

通过JdbcTemplate 实现批处理需要实现特定接口的两个方法，BatchPreparedStatementSetter，并且将其作为第二个参数传入到batchUpdate方法调用中。使用getBatchSize提供当前批量操作的大小。使用setValues方法设置语句的Value参数。这个方法会按getBatchSize设置中指定的调用次数。下面的例子中通过传入列表来批量更新actor表。在这个例子中整个列表使用了批量操作：

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        int[] updateCounts = jdbcTemplate.batchUpdate("update t_
actor set first_name = ?, " +
                "last_name = ? where id = ?",
            new BatchPreparedStatementSetter() {
                public void setValues(PreparedStatement ps, int
i) throws SQLException {
                        ps.setString(1, actors.get(i).getFirstNa
me());
                        ps.setString(2, actors.get(i).getLastNam
e());
                        ps.setLong(3, actors.get(i).getId().long
Value());
                    }

                    public int getBatchSize() {
                        return actors.size();
                    }
                });
        return updateCounts;
    }

    // ... additional methods
}
```

如果你需要处理批量更新或者从文件中批量读取，你可能需要确定一个合适的批处理大小，但是最后一次批处理可能达不到这个大小。在这种场景下你可以使用InterruptibleBatchPreparedStatementSetter接口，允许在输入流耗尽之后终止批处理，isBatchExhausted方法使得你可以指定批处理结束时间。

15.4.2 对象列表的批量处理

JdbcTemplate和NamedParameterJdbcTemplate都提供了批量更新的替代方案。这个时候不是实现一个特定的批量接口，而是在调用时传入所有的值列表。框架会循环访问这些值并且使用内部的SQL语句setter方法。你是否已声明参数对应API是不一样的。针对已声明参数你需要传入qlParameterSource数组，每项对应单次的批量操作。你可以使用SqlParameterSource.createBatch方法来创建这个数组，传入JavaBean数组或是包含参数值的Map数组。

下面是一个使用已声明参数的批量更新例子：

```
public class JdbcActorDao implements ActorDao {
    private NamedParameterTemplate namedParameterJdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.namedParameterJdbcTemplate = new NamedParameterJdbc
Template(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        SqlParameterSource[] batch = SqlParameterSourceUtils.cre
ateBatch(actors.toArray());
        int[] updateCounts = namedParameterJdbcTemplate.batchUpd
ate(
                "update t_actor set first_name = :firstName, las
t_name = :lastName where id = :id",
                batch);
        return updateCounts;
    }

    // ... additional methods
}
```

对于使用“？”占位符的SQL语句，你需要传入带有更新值的对象数组。对象数组每一项对应SQL语句中的一个占位符，并且传入顺序需要和SQL语句中定义的顺序保持一致。

下面是使用经典JDBC“？”占位符的例子：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        List<Object[]> batch = new ArrayList<Object[]>();
        for (Actor actor : actors) {
            Object[] values = new Object[] {
                    actor.getFirstName(),
                    actor.getLastName(),
                    actor.getId()};
            batch.add(values);
        }
        int[] updateCounts = jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name =
? where id = ?",
                batch);
        return updateCounts;
    }

    // ... additional methods

}
```

上面所有的批量更新方法都返回一个数组，包含具体成功的行数。这个计数是由JDBC驱动返回的。如果拿不到计数。JDBC驱动会返回-2。

15.4.3 多个批处理操作

上面最后一个例子更新的批处理数量太大，最好能再分割成更小的块。最简单的方式就是你多次调用batchUpdate来实现，但是可以有更优的方法。要使用这个方法除了SQL语句，还需要传入参数集合对象，每次Batch的更新数和一个ParameterizedPreparedStatementSetter去设置预编译SQL语句的参数值。框架会循环调用提供的值并且将更新操作切割成指定数量的小批次。

下面的例子设置了更新批次数量为100的批量更新操作：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[][] batchUpdate(final Collection<Actor> actors) {
        int[][] updateCounts = jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name =
? where id = ?",
                actors,
                100,
                new ParameterizedPreparedStatementSetter<Actor>(
) {
                    public void setValues(PreparedStatement ps,
Actor argument) throws SQLException {
                        ps.setString(1, argument.getFirstName())
;
                        ps.setString(2, argument.getLastName());
                        ps.setLong(3, argument.getId().longValue
());
                    }
                });
        return updateCounts;
    }

    // ... additional methods

}
```

这个调用的批量更新方法返回一个包含int数组的二维数组，包含每次更新生效的行数。第一层数组长度代表批处理执行的数量，第二层数组长度代表每个批处理生效的更新数。每个批处理的更新数必须和所有批处理的大小匹配，除非是最后一次批处理可能小于这个数，具体依赖于更新对象的总数。每次更新语句生效的更新数由JDBC驱动提供。如果更新数量不存在，JDBC驱动会返回-2

15.5 利用SimpleJdbc类简化JDBC操作

SimpleJdbcInsert类和SimpleJdbcCall类主要利用了JDBC驱动所提供的数据库元数据的一些特性来简化数据库操作配置。这意味着可以在前端减少配置，当然你也可以覆盖或是关闭底层的元数据处理，在代码里面指定所有的细节。

15.5.1 利用SimpleJdbcInsert插入数据

让我们首先看SimpleJdbcInsert类可提供的最小配置选项。你需要在数据访问层初始化方法里面初始化SimpleJdbcInsert类。在这个例子中，初始化方法是setDataSource。你不需要继承SimpleJdbcInsert，只需要简单的创建其实例同时调用withTableName设置数据库名。

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource).with
TableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Obj
ect>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

代码中的execute只传入java.utils.Map作为唯一参数。需要注意的是Map里面用到的Key必须和数据库中表对应的列名一一匹配。这是因为我们需要按顺序读取元数据来构造实际的插入语句。

15.5.2 使用SimpleJdbcInsert获取自增Key

接下来，我们对于同样的插入语句，我们并不传入id，而是通过数据库自动获取主键的方式来创建新的Actor对象并插入数据库。 当我们创建SimpleJdbcInsert实例时, 我们不仅需要指定表名，同时我们通过usingGeneratedKeyColumns方法指定需要数据库自动生成主键的列名。

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Obj
ect>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameter
s);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

执行插入操作时第二种方式最大的区别是你不是在Map中指定ID，而是调用executeAndReturnKey方法。这个方法返回java.lang.Number对象，可以创建一个数值类型的实例用于我们的领域模型中。你不能仅仅依赖所有的数据库都返回一个指定的Java类；java.lang.Number是你可以依赖的基础类。如果你有多个自增列，或者自增的值是非数值型的，你可以使用executeAndReturnKeyHolder 方法返回的KeyHolder

15.5.3 使用SimpleJdbcInsert指定列

你可以在插入操作中使用usingColumns方法来指定特定的列名

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingColumns("first_name", "last_name")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Obj
ect>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameter
s);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

这里插入操作的执行和你依赖元数据决定更新哪个列的方式是一样的。

15.5.4 使用SqlParameterSource 提供参数值

使用Map来指定参数值没有问题，但不是最便捷的方法。Spring提供了一些SqlParameterSource接口的实现类来更方便的做这些操作。
第一个是BeanPropertySqlParameterSource，如果你有一个JavaBean兼容的类包含具体的值，使用这个类是很方便的。他会使用相关的Getter方法来获取参数值。下面是一个例子：

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new BeanPropertySqlParam
eterSource(actor);
        Number newId = insertActor.executeAndReturnKey(parameter
s);
        actor.setId(newId.longValue());
    }

    // ... additional methods

}
```

另外一个选择是使用MapSqlParameterSource，类似于Map、但是提供了一个更便捷的addValue方法可以用来做链式操作。

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new MapSqlParameterSourc
e()
                .addValue("first_name", actor.getFirstName())
                .addValue("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameter
s);
        actor.setId(newId.longValue());
    }

    // ... additional methods

}
```

上面这些例子可以看出、配置是一样的，区别只是切换了不同的提供参数的实现方式来执行调用。

15.5.5 利用SimpleJdbcCall调用存储过程

SimpleJdbcCall利用数据库元数据的特性来查找传入的参数和返回值，这样你就不需要显式去定义他们。如果你喜欢的话也以自己定义参数，尤其对于某些参数，你无法直接将他们映射到Java类上，例如ARRAY类型和STRUCT类型的参数。下面第一个例子展示了一个存储过程，从一个MySQL数据库返回Varchar和Date类型。这个存储过程例子从指定的actor记录中查询返回first_name,last_name,和birth_date列。

```
CREATE PROCEDURE read_actor (
    IN in_id INTEGER,
    OUT out_first_name VARCHAR(100),
    OUT out_last_name VARCHAR(100),
    OUT out_birth_date DATE)
BEGIN
    SELECT first_name, last_name, birth_date
    INTO out_first_name, out_last_name, out_birth_date
    FROM t_actor where id = in_id;
END;
```

in_id 参数包含你正在查找的actor记录的id.out参数返回从数据库表读取的数据

SimpleJdbcCall 和SimpleJdbcInsert定义的方式比较类似。你需要在数据访问层的初始化代码中初始化和配置该类。相比StoredProcedure类，你不需要创建一个子类并且不需要定义能够在数据库元数据中查找到的参数。下面是一个使用上面存储过程的SimpleJdbcCall配置例子。除了DataSource以外唯一的配置选项是存储过程的名字

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.procReadActor = new SimpleJdbcCall(dataSource)
                .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }

    // ... additional methods

}
```

调用代码包括创建包含传入参数的SqlParameterSource。这里需要重视的是传入参数值名字需要和存储过程中定义的参数名称相匹配。有一种场景不需要匹配、那就是你使用元数据去确定数据库对象如何与存储过程相关联。在存储过程源代码中指定的并不一定是数据库中存储的格式。有些数据库会把名字转成大写、而另外一些会使用小写或者特定的格式。

execute方法接受传入参数，同时返回一个Map包含任意的返回参数，Map的Key是存储过程中指定的名字。在这个例子中它们是out_first_name, out_last_name 和 out_birth_date

execute 方法的最后一部分使用返回的数据创建Actor对象实例。再次需要强调的是Out参数的名字必须是存储过程中定义的。结果Map中存储的返回参数名必须和数据库中的返回参数名（不同的数据库可能会不一样）相匹配，为了提高你代码的可

重用性，你需要在查找中区分大小写，或者使用Spring里面的LinkedCaseInsensitiveMap。如果使用LinkedCaseInsensitiveMap，你需要创建自己的JdbcTemplate并且将setResultsMapCaseInsensitive属性设置为True。然后你将自定义的JdbcTemplate 传入到SimpleJdbcCall的构造器中。下面是这种配置的一个例子：

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
;
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_actor");
    }

    // ... additional methods

}
```

通过这样的配置，你就可以无需担心返回参数值的大小写问题。

15.5.6 为SimpleJdbcCall显式定义参数

你已经了解如何通过元数据来简化参数配置，但如果你需要的话也可以显式指定参数。这样做的方法是在创建SimpleJdbcCall类同时通过declareParameters方法进行配置，这个方式可以传入一系列的SqlParameter。下面的章节会详细描述如何定义一个SqlParameter

> 备注：如果你使用的数据库不是Spring支持的数据库类型的话显式定义就很有必要了。当前Spring支持以下数据库的存储过程元数据查找能力：Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, 和 Sybase. 我们同时对某些数据库内置函数支持元数据特性：比如：MySQL、Microsoft SQL Server和Oracle。

你可以选择显式定义一个、多个，或者所有参数。当你没有显式定义参数时元数据参数仍然会被使用。当你不想用元数据查找参数功能、只想指定参数时，需要调用withoutProcedureColumnMetaDataAccess方法。假设你针对同一个数据函数定义了两个或多个不同的调用方法签名，在每一个给定的签名中你需要使用useInParameterNames来指定传入参数的名称列表。下面是一个完全自定义的存储过程调用例子

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
;
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_actor")
                .withoutProcedureColumnMetaDataAccess()
                .useInParameterNames("in_id")
                .declareParameters(
                        new SqlParameter("in_id", Types.NUMERIC)
,
                        new SqlOutParameter("out_first_name", Ty
pes.VARCHAR),
                        new SqlOutParameter("out_last_name", Typ
es.VARCHAR),
                        new SqlOutParameter("out_birth_date", Ty
pes.DATE)
                );
    }

    // ... additional methods
}
```

两个例子的执行结果是一样的，区别是这个例子显式指定了所有细节，而不是仅仅依赖于数据库元数据。

15.5.7 如何定义SqlParameters

如何定义SimpleJdbc类和RDBMS操作类的参数，详见15.6：“像Java对象那样操作JDBC”,
你需要使用SqlParameter或者是它的子类。通常需要在构造器中定义参数名和SQL类型。SQL类型使用java.sql.Types常量来定义。
我们已经看到过类似于如下的定义：

```
new SqlParameter("in_id", Types.NUMERIC),
    new SqlOutParameter("out_first_name", Types.VARCHAR),
```

上面第一行SqlParameter 定义了一个传入参数。IN参数可以同时在存储过程调用和SqlQuery查询中使用，它的子类在下面的章节也有覆盖。

上面第二行SqlOutParameter定义了在一次存储过程调用中使用的返回参数。还有一个SqlInOutParameter类，可以用于输入输出参数。也就是说，它既是一个传入参数，也是一个返回值。

> 备注：参数只有被定义成SqlParameter和SqlInOutParameter才可以提供输入值。不像StoredProcedure类为了考虑向后兼容允许定义为SqlOutParameter的参数可以提供输入值

对于输入参数，除了名字和SQL类型，你可以定义数值区间或是自定义数据类型名。针对输出参数，你可以使用RowMapper处理从REF游标返回的行映射。另外一种选择是定义SqlReturnType，可以针对返回值作自定义处理。

15.5.8 使用SimpleJdbcCall调用内置存储函数

调用存储函数几乎和调用存储过程的方式是一样的，唯一的区别你提供的是函数名而不是存储过程名。你可以使用withFunctionName方法作为配置的一部分表示我们想要调用一个函数，以及生成函数调用相关的字符串。一个特殊的execute调用，executeFunction，用来指定这个函数并且返回一个指定类型的函数值，这意味着你不需要从结果Map获取返回值。存储过程也有一个名字为executeObject的便捷方法，但是只要一个输出参数。下面的例子基于一个名字为get_actor_name的存储函数，返回actor的全名。下面是这个函数的Mysql源代码：

```
CREATE FUNCTION get_actor_name (in_id INTEGER)
RETURNS VARCHAR(200) READS SQL DATA
BEGIN
    DECLARE out_name VARCHAR(200);
    SELECT concat(first_name, ' ', last_name)
        INTO out_name
        FROM t_actor where id = in_id;
    RETURN out_name;
END;
```

我们需要在初始方法中创建SimpleJdbcCall来调用这个函数

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
;
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)
                .withFunctionName("get_actor_name");
    }

    public String getActorName(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        String name = funcGetActorName.executeFunction(String.cl
ass, in);
        return name;
    }

    // ... additional methods

}
```

execute方法返回一个包含函数调用返回值的字符串

15.5.9 从SimpleJdbcCall返回ResultSet/REF游标

调用存储过程或者函数返回结果集会相对棘手一点。一些数据库会在JDBC结果处理中返回结果集，而另外一些数据库则需要明确指定返回值的类型。两种方式都需要循环迭代结果集做额外处理。通过SimpleJdbcCall，你可以使用returningResultSet方法，并定义一个RowMapper的实现类来处理特定的返回值。当结果集在返回结果处理过程中没有被定义名称时，返回的结果集必须与定义的RowMapper的实现类指定的顺序保持一致。 而指定的名字也会被用作返回结果集中的名称。

下面的例子使用了一个不包含输入参数的存储过程并且返回t_actor标的所有行。下面是这个存储过程的Mysql源代码：

```
CREATE PROCEDURE read_all_actors()
BEGIN
 SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_act
or a;
END;
```

调用这个存储过程你需要定义RowMapper。因为我们定义的Map类遵循JavaBean规范，所以我们可以使用BeanPropertyRowMapper作为实现类。 通过将相应的class类作为参数传入到newInstance方法中，我们可以创建这个实现类。

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadAllActors;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
;
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadAllActors = new SimpleJdbcCall(jdbcTemplate
)
                .withProcedureName("read_all_actors")
                .returningResultSet("actors",
                BeanPropertyRowMapper.newInstance(Actor.class));
    }

    public List getActorsList() {
        Map m = procReadAllActors.execute(new HashMap<String, Ob
ject>(0));
        return (List) m.get("actors");
    }

    // ... additional methods

}
```

execute调用传入一个空Map，因为这里不需要传入任何参数。从结果Map中提取Actors列表，并且返回给调用者。

15.6 像Java对象那样操作JDBC

org.springframework.jdbc.object包能让你更加面向对象化的访问数据库。举个例子，用户可以执行查询并返回一个list， 该list作为一个结果集将把从数据库中取出的列数据映射到业务对象的属性上。你也可以执行存储过程，包括更新、删除、插入语句。

备注：许多Spring的开发者认为下面将描述的各种RDBMS操作类（StoredProcedure类除外）可以直接被JdbcTemplate代替； 相对于把一个查询操作封装成一个类而言，直接调用JdbcTemplate方法将更简单而且更容易理解。但这仅仅是一种观点而已， 如果你认为可以从直接使用RDBMS操作类中获取一些额外的好处，你不妨根据自己的需要和喜好进行不同的选择。

15.6.1 SqlQuery

SqlQuery类主要封装了SQL查询，本身可重用并且是线程安全的。子类必须实现newRowMapper方法，这个方法提供了一个RowMapper实例，用于在查询执行返回时创建的结果集迭代过程中每一行映射并创建一个对象。SqlQuery类一般不会直接使用；因为MappingSqlQuery子类已经提供了一个更方便从列映射到Java类的实现。其他继承SqlQuery的子类有MappingSqlQueryWithParameters和UpdatableSqlQuery。

15.6.2 MappingSqlQuery

MappingSqlQuery是一个可重用的查询类，它的子类必须实现mapRow(..)方法，将结果集返回的每一行转换成指定的对象类型。下面的例子展示了一个自定义的查询例子，将t_actor关系表的数据映射成Actor类。

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {

    public ActorMappingQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from t_actor
 where id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTE
GER));
        compile();
    }

    @Override
    protected Actor mapRow(ResultSet rs, int rowNumber) throws S
QLException {
        Actor actor = new Actor();
        actor.setId(rs.getLong("id"));
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }

}
```

这个类继承了MappingSqlQuery，并且传入Actor类型的泛型参数。这个自定义查询类的构造函数将DataSource作为唯一的传入参数。这个构造器中你调用父类的构造器，传入DataSource以及相应的SQL参数。该SQL用于创建PreparedStatement，因此它可能包含任何在执行过程中传入参数的占位符。你必须在SqlParameter中使用declareParameter方法定义每个参数。SqlParameter使用java.sql.Types定义名字和JDBC类型。在你定义了所有的参数后，你需要调用compile方法，语句被预编译后方便后续的执行。这个类在编译后是线程安全的，一旦在DAO初始化时这些实例被创建后，它们可以作为实例变量一直被重用。

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

这个例子中的方法通过唯一的传入参数id获取customer实例。因为我们只需要返回一个对象，所以就简单的调用findObject类就可以了，这个方法只需要传入id参数。如果我们需要一次查询返回一个列表的话，就需要使用传入可变参数数组的执行方法。

```
public List<Actor> searchForActors(int age, String namePattern)
{
    List<Actor> actors = actorSearchMappingQuery.execute(age, na
mePattern);
    return actors;
}
```

15.6.3 SqlUpdate

SqlUpdate封装了SQL的更新操作。和查询一样，更新对象是可以被重用的，就像所有的rdbms操作类，更新操作能够传入参数并且在SQL定义。类似于SqlQuery诸多execute(..)方法，这个类提供了一系列update(..)方法。SQLUpdate类不是抽象类，它可以被继承，比如，实现自定义的更新方法。但是你并不需要继承SqlUpdate类来达到这个目的，你可以更简单的在SQL中设置自定义参数来实现。

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id =
 ?");
        declareParameter(new SqlParameter("creditRating", Types.
NUMERIC));
        declareParameter(new SqlParameter("id", Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int execute(int id, int rating) {
        return update(rating, id);
    }
}
```

15.6.4 StoredProcedure

StoredProcedure类是所有RDBMS存储过程的抽象类。该类提供了多种execute(..)方法，其访问类型都是protected的。

为了定义一个存储过程类，你需要使用SqlParameter或者它的一个子类。你必须像下面的代码例子那样在构造函数中指定参数名和SQL类型。SQL类型使用java.sql.Types 常量定义。

```
new SqlParameter("in_id", Types.NUMERIC),
    new SqlOutParameter("out_first_name", Types.VARCHAR),
```

SqlParameter的第一行定义了一个输入参数。输入参数可以同时被存储过程调用和使用SqlQuery的查询语句使用，他的子类会在下面的章节提到。

第二行SqlOutParameter 参数定义了一个在存储过程调用中使用的输出参数。SqlInOutParameter 还有一个InOut参数，该参数提供了一个输入值，同时也有返回值。

对应输入参数，除了名字和SQL类型，你还能指定返回区间数值类型和自定义数据库类型。对于输出参数你可以使用RowMapper来处理REF游标返回的行映射关系。另一个选择是指定SqlReturnType，能够让你定义自定义的返回值类型。

下面的程序演示了如何调用Oracle中的sysdate()函数。为了使用存储过程函数你需要创建一个StoredProcedure的子类。在这个例子中，StoredProcedure是一个内部类，但是如果你需要重用StoredProcedure你需要定义成一个顶级类。这个例子没有输入参数，但是使用SqlOutParameter类定义了一个时间类型的输出参数。execute（）方法执行了存储过程，并且从结果集Map中获取返回的时间数据。结果集Map中包含每个输出参数对应的项,在这个例子中就只有一项，使用了参数名作为key.

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class StoredProcedureDao {

    private GetSysdateProcedure getSysdate;

    @Autowired
```

```
    public void init(DataSource dataSource) {
        this.getSysdate = new GetSysdateProcedure(dataSource);
    }

    public Date getSysdate() {
        return getSysdate.execute();
    }

    private class GetSysdateProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public GetSysdateProcedure(DataSource dataSource) {
            setDataSource(dataSource);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.D
ATE));
            compile();
        }

        public Date execute() {
            // the 'sysdate' sproc has no input parameters, so a
n empty Map is supplied...
            Map<String, Object> results = execute(new HashMap<St
ring, Object>());
            Date sysdate = (Date) results.get("date");
            return sysdate;
        }
    }

}
```

下面是一个包含两个输出参数的存储过程例子。

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProced
ure {

    private static final String SPROC_NAME = "AllTitlesAndGenres
";

    public TitlesAndGenresStoredProcedure(DataSource dataSource)
 {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTyp
es.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTyp
es.CURSOR, new GenreMapper()));
        compile();
    }

    public Map<String, Object> execute() {
        // again, this sproc has no input parameters, so an empt
y Map is supplied
        return super.execute(new HashMap<String, Object>());
    }
}
```

值得注意的是TitlesAndGenresStoredProcedure构造函数中 declareParameter(..)的SqlOutParameter参数， 该参数使用RowMapper接口的实现。这是一种非常方便有效的重用方式。两种RowMapper实现的代码如下：

TitleMapper类将返回结果集的每一行映射成Title类

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Title;

public final class TitleMapper implements RowMapper<Title> {

    public Title mapRow(ResultSet rs, int rowNum) throws SQLExce
ption {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

GenreMapper类将返回结果集的每一行映射成Genre类

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper<Genre> {

    public Genre mapRow(ResultSet rs, int rowNum) throws SQLExce
ption {
        return new Genre(rs.getString("name"));
    }
}
```

为了将参数传递给RDBMS中定义的一个或多个输入参数给存储过程，你可以定义一个强类型的execute(..)方法，该方法将调用基类的protected execute(Map parameters)方法。例如：

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;

import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProced
ure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate"
;

    public TitlesAfterDateStoredProcedure(DataSource dataSource)
 {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Typ
es.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTyp
es.CURSOR, new TitleMapper()));
        compile();
    }

    public Map<String, Object> execute(Date cutoffDate) {
        Map<String, Object> inputs = new HashMap<String, Object>
();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

15.7 参数和数据处理的常见问题

Spring JDBC框架提供了多个方法来处理常见的参数和数据问题

15.7.1 为参数设置SQL的类型信息

通常Sping通过传入的参数类型决定SQL的参数类型。可以在设定参数值的时候显式提供SQL类型。有些场景下设置NULL值是有必要的。

你可以通过以下方式来设置SQL类型信息：

许多JdbcTemplate的更新和查询方法需要传入额外的int数组类型的参数。这个数组使用java.sql.Types类的常量值来确定相关参数的SQL类型。每个参数会有对应的类型项。

你可以使用SqlParameterValue类来包装需要额外信息的参数值。针对每个值创建一个新的实例，并且在构造函数中传入SQL类型和参数值。你还可以传入数值类型的可选区间参数

对于那些使用命名参数的情况，使用SqlParameterSource类型的类比如BeanPropertySqlParameterSource ，或MapSqlParameterSource。他们都具备了为命名参数注册SQL类型的功能。

15.7.2 处理BLOB和CLOB对象

你可以存储图片，其他类型的二进制数据，和数据库里面的大块文本。这些大的对象叫做BLOBS（全称：Binary Large OBject；用于二进制数据）和CLOBS（全称：Character Large OBject；用于字符数据）。在Spring中你可以使用JdbcTemplate直接处理这些大对象，并且也可以使用RDBMS对象提供的上层抽象类，或者使用SimpleJdbc类。所有这些方法使用LobHandler接口的实现类来处理LOB数据的管理（全称：Large Object）。LobHandler通过getLobCreator方法提供了对LobCreator 类的访问，用于创建新的LOB插入对象。

LobCreator/LobHandler提供了LOB输入和输出的支持：

```
BLOB：
  byte[] — getBlobAsBytes和setBlobAsBytes
  InputStream - getBlobAsBinaryStream和setBlobAsBinaryStream

CLOB：
  String - getClobAsString和setClobAsString
  InputStream - getClobAsAsciiStream和setClobAsAsciiStream
  Reader - getClobAsCharacterStream和setClobAsCharacterStream
```

下面的例子展示了如何创建和插入一个BLOB。后面的例子我们将举例如何从数据库中将BLOB数据读取出来

这个例子使用了JdbcTemplate和AbstractLobCreatingPreparedStatementCallback的实现类。它主要实现了一个方法，setValues.这个方法提供了用于在你的SQL插入语句中设置LOB列的LobCreator。

针对这个例子我们假定有一个变量lobHandler，已经设置了DefaultLobHandler的一个实例。通常你可以使用依赖注入来设置这个值。

```
final File blobIn = new File("spring2004.jpg");
final InputStream blobIs = new FileInputStream(blobIn);
final File clobIn = new File("large.txt");
final InputStream clobIs = new FileInputStream(clobIn);
final InputStreamReader clobReader = new InputStreamReader(clobI
s);
jdbcTemplate.execute(
    "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)
",
    new AbstractLobCreatingPreparedStatementCallback(lobHandler)
 { （1）
        protected void setValues(PreparedStatement ps, LobCreato
r lobCreator) throws SQLException {
            ps.setLong(1, 1L);
            lobCreator.setClobAsCharacterStream(ps, 2, clobReade
r, (int)clobIn.length()); （2）
            lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int
)blobIn.length()); （3）
        }
    }
);
blobIs.close();
clobReader.close();
```

> （1）、这里例子中传入的lobHandler 使用了默认实现类DefaultLobHandler
>
> （2）、使用setClobAsCharacterStream传入CLOB的内容
>
> （3）、使用setBlobAsBinaryStream传入BLOB的内容
>
> 备注：如果你调用从DefaultLobHandler.getLobCreator()返回的LobCreator的setBlobAsBinaryStream, setClobAsAsciiStream, 或者setClobAsCharacterStream方法，其中contentLength参数允许传入一个负值。如果指定的内容长度是负值，DefaultLobHandler会使用JDBC4.0不带长度参数的set-stream方法，或者直接传入驱动指定的长度；JDBC驱动对未指定长度的LOB流的支持请参见相关文档

下面是从数据库读取LOB数据的例子。我们这里再次使用JdbcTempate并使用相同的DefaultLobHandler实例。

```
List<Map<String, Object>> l = jdbcTemplate.query("select id, a_c
lob, a_blob from lob_table",
    new RowMapper<Map<String, Object>>() {
        public Map<String, Object> mapRow(ResultSet rs, int i) t
hrows SQLException {
            Map<String, Object> results = new HashMap<String, Ob
ject>();
            String clobText = lobHandler.getClobAsString(rs, "a_
clob");   1
results.put("CLOB", clobText); byte[] blobBytes = lobHandler.get
BlobAsBytes(rs, "a_blob");    2
results.put("BLOB", blobBytes); return results; } });
```

> 1、使用getClobAsString获取CLOB的内容
>
> 2、使用getBlobAsBytes获取BLOC的内容

15.7.3 传入IN语句的列表值

SQL标准允许基于一个带参数列表的表达式进行查询，一个典型的例子是select * from T_ACTOR where id in (1, 2, 3). 这样的可变参数列表没有被JDBC标准直接支持；你不能定义可变数量的占位符（placeholder），只能定义固定变量的占位符，或者你在动态生成SQL字符串的时候需要提前知晓所需占位符的数量。NamedParameterJdbcTemplate 和 JdbcTemplate 都使用了后面那种方式。当你传入参数时，你需要传入一个java.util.List类型，支持基本类型。而这个list将会在SQL执行时替换占位符并传入参数。

> 备注：在传入多个值的时候需要注意。JDBC标准不保证你能在一个in表达式列表中传入超过100个值，不少数据库会超过这个值，但是一般都会有个上限。比如Oracle的上限是1000.

除了值列表的元数据值，你可以创建java.util.List的对象数组。这个列表支持多个在in语句内定义的表达式例如
select * from T_ACTOR where (id, last_name) in ((1, ‘Johnson’), (2, ‘Harrop’\))。当然有个前提你的数据库需要支持这个语法。

15.7.4 处理存储过程调用的复杂类型

当你调用存储过程时有时需要使用数据库特定的复杂类型。为了兼容这些类型，当存储过程调用返回时Spring提供了一个SqlReturnType 来处理这些类型，SqlTypeValue用于存储过程的传入参数。

下面是一个用户自定义类型ITEM_TYPE的Oracle STRUCT对象的返回值例子。SqlReturnType有一个方法getTypeValue必须被实现。而这个接口的实现将被用作SqlOutParameter声明的一部分。

```
final TestItem = new TestItem(123L, "A test item",
        new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT,
 "ITEM_TYPE",
    new SqlReturnType() {
        public Object getTypeValue(CallableStatement cs, int col
Indx, int sqlType, String typeName) throws SQLException {
            STRUCT struct = (STRUCT) cs.getObject(colIndx);
            Object[] attr = struct.getAttributes();
            TestItem item = new TestItem();
            item.setId(((Number) attr[0]).longValue());
            item.setDescription((String) attr[1]);
            item.setExpirationDate((java.util.Date) attr[2]);
            return item;
        }
    }));
```

你可以使用SqlTypeValue 类往存储过程传入像TestItem那样的Java对象。你必须实现SqlTypeValue接口的createTypeValue方法。你可以使用传入的连接来创建像StructDescriptors这样的数据库指定对象。下面是相关的例子。

```
SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlTyp
e, String typeName) throws SQLException {
        StructDescriptor itemDescriptor = new StructDescriptor(t
ypeName, conn);
        Struct item = new STRUCT(itemDescriptor, conn,
        new Object[] {
            testItem.getId(),
            testItem.getDescription(),
            new java.sql.Date(testItem.getExpirationDate().getTi
me())
        });
        return item;
    }
};
```

SqlTypeValue会加入到包含输入参数的Map中，用于执行存储过程调用。

SqlTypeValue 的另外一个用法是给Oracle的存储过程传入一个数组。Oracle内部有它自己的ARRAY类，在这些例子中一定会被使用，你可以使用SqlTypeValue来创建Oracle ARRAY的实例，并且设置到Java ARRAY类的值中。

```
final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlTyp
e, String typeName) throws SQLException {
        ArrayDescriptor arrayDescriptor = new ArrayDescriptor(ty
peName, conn);
        ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
        return idArray;
    }
};
```

15.8 内嵌数据库支持

org.springframework.jdbc.datasource.embedded包包含对内嵌Java数据库引擎的支持。如对HSQL, H2, and Derby原生支持，你还可以使用扩展API来嵌入新的数据库内嵌类型和Datasource实现。

15.8.1 为什么使用一个内嵌数据库？

内嵌数据库因为比较轻量级所以在开发阶段比较方便有用。包括配置比较容易，启动快，方便测试，并且在开发阶段方便快速设计SQL操作

15.8.2 使用Spring配置来创建内嵌数据库

如果你想要将内嵌的数据库实例作为Bean配置到Spring的ApplicationContext中，使用spring-jdbc命名空间下的embedded-database tag

```
<jdbc:embedded-database id="dataSource" generate-name="true">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

上面的配置创建了一个内嵌的HSQL数据库，并且在classpath下面配置schema.sql和test-data.sql资源。同时，作为一种最佳实践，内嵌数据库会被制定一个唯一生成的名字。内嵌数据库在Spring容器中作为javax.sql.DataSource Bean类型存在，并且能够被注入到所需的数据库访问对象中。

15.8.3 使用编程方式创建内嵌数据库

EmbeddedDatabaseBuilder提供了创建内嵌数据库的流式API。当你在独立的环境中或者是在独立的集成测试中可以使用这种方法创建一个内嵌数据库，下面是一个例子：

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
    .generateUniqueName(true)
    .setType(H2)
    .setScriptEncoding("UTF-8")
    .ignoreFailedDrops(true)
    .addScript("schema.sql")
    .addScripts("user_data.sql", "country_data.sql")
    .build();

// perform actions against the db (EmbeddedDatabase extends java
x.sql.DataSource)

db.shutdown()
```

更多支持的细节请参见：EmbeddedDatabaseBuilder 的JavaDoc

EmbeddedDatabaseBuilder 也可以使用Java Config类来创建内嵌数据库，下面是一个例子：

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .generateUniqueName(true)
            .setType(H2)
            .setScriptEncoding("UTF-8")
            .ignoreFailedDrops(true)
            .addScript("schema.sql")
            .addScripts("user_data.sql", "country_data.sql")
            .build();
    }
}
```

15.8.4 选择内嵌数据库的类型

使用**HSQL**

spring支持HSQL 1.8.0及以上版本。HSQL是缺省默认内嵌数据库类型。如果显式指定HSQL，设置embedded-database Tag的type属性值为HSQL。如果使用builderAPI.调用EmbeddedDatabaseType.HSQL的setType(EmbeddedDatabaseType)方法。

使用**H2**

Spring也支持H2数据库。设置embedded-database tag的type类型值为H2来启用H2。如果你使用了builder API。调用setType(EmbeddedDatabaseType) 方法设置值为EmbeddedDatabaseType.H2。

使用**Derby**

Spring也支持 Apache Derby 10.5及以上版本，设置embedded-database tag的type属性值为BERBY来开启DERBY。如果你使用builder API，调用setType(EmbeddedDatabaseType)方法设置值为EmbeddedDatabaseType.DERBY.

15.8.5 使用内嵌数据库测试数据访问层逻辑

内嵌数据库提供了数据访问层代码的轻量级测试方案，下面是使用了内嵌数据库的数据访问层集成测试模板。使用这样的模板当内嵌数据库不需要在测试类中被重用时是有用的。不过，当你希望创建可以在test集中共享的内嵌数据库。考虑使用Spring TestContext测试框架，同时在Spring ApplicationContext中将内嵌数据库配置成一个Bean，具体参见15.8.2节, “使用Spring配置来创建内嵌数据库” 和15.8.3节, “使用编程方式创建内嵌数据库”.

```
public class DataAccessIntegrationTestTemplate {

    private EmbeddedDatabase db;

    @Before
    public void setUp() {
        // creates an HSQL in-memory database populated from def
ault scripts
        // classpath:schema.sql and classpath:data.sql
        db = new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .addDefaultScripts()
                .build();
    }

    @Test
    public void testDataAccess() {
        JdbcTemplate template = new JdbcTemplate(db);
        template.query( /* ... */ );
    }

    @After
    public void tearDown() {
        db.shutdown();
    }

}
```

15.8.6 生成内嵌数据库的唯一名字

开发团队在使用内嵌数据库时经常碰到的一个错误是：当他们的测试集想对同一个数据库创建额外的实例。这种错误在以下场景经常发生，XML配置文件或者@Configuration类用于创建内嵌数据库，并且相关的配置在同样的测试集的多个测试场景下都被用到（例如，在同一个JVM进程中）。例如，针对内嵌数据库的不同集成测试的ApplicationContext配置的区别只在当前哪个Bean定义是有效的。

这些错误的根源是Spring的EmbeddedDatabaseFactory工厂（ XML命名空间和Java Config对象的EmbeddedDatabaseBuilder内部都用到了这个）会将内嵌数据库的名字默认设置成”testdb”.针对的场景，内嵌数据库通常设置成和Bean Id相同的名字。（例如，常用像“dataSource”的名字）。结果，接下来创建内嵌数据库的尝试都没创建一个新的数据库。相反，同样的JDBC链接URL被重用。创建内嵌数据的库的尝试往往从同一个配置返回了已存在的内嵌数据库实例。

为了解决这个问题Spring框架4.2 提供了生成内嵌数据库唯一名的支持。例如：

```
EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName()
EmbeddedDatabaseBuilder.generateUniqueName()
<jdbc:embedded-database generate-name="true" … >
```

15.8.7 内嵌数据库扩展支持

Spring JDBC 内嵌数据库支持以下两种扩展支持：

实现EmbeddedDatabaseConfigurer支持新的内嵌数据库类型。

实现DataSourceFactory支持新的DataSource实现，例如管理内嵌数据库连接的连接池

欢迎贡献内部扩展给Spring社区，相关网址见：jira.spring.io.

15.9 初始化Datasource

org.springframework.jdbc.datasource.init用于支持初始化一个现有的DataSource。内嵌数据库提供了创建和初始化Datasource的一个选项，但是有时你需要在另外的服务器上初始化实例。

15.9.1 使用Spring XML来初始化数据库

如果你想要初始化一个数据库你可以设置DataSource Bean的引用，使用spring-jdbc命名空间下的initialize-database标记

```
<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql"/
>
    <jdbc:script location="classpath:com/foo/sql/db-test-data.sq
l"/>
</jdbc:initialize-database>
```

上面的例子执行了数据库的两个脚本：第一个脚本创建了一个Schema，第二个往表里插入了一个测试数据集。脚本路径可以使用Spring中ant类型的查找资源的模式（例如classpath*:/com/foo/**/sql/*-data.sql）。如果使用了正则表达式，脚本会按照URL或者文件名的词法顺序执行。

默认数据库初始器会无条件执行该脚本。有时你并不想要这么做，例如。你正在执行一个已存在测试数据集的数据库脚本。下面的通用方法会避免不小心删除数据，比如像上面的例子先创建表然后再插入-第一步在表已经存在时会失败掉。

为了能够在创建和删除已有数据方面提供更多的控制，XML命名空间提供了更多的方式。第一个是将初始化设置开启还是关闭。这个可以根据当前环境情况设置（例如用系统变量或者环境属性Bean中获取一个布尔值）例如：

```
<jdbc:initialize-database data-source="dataSource"
    enabled="#{systemProperties.INITIALIZE_DATABASE}">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

第二个选项是控制当前数据的行为，这是为了提高容错性。你能够控制初始器来忽略SQL里面执行脚本的特定错误、例如：

```
<jdbc:initialize-database data-source="dataSource" ignore-failur
es="DROPS">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

在这个例子中我们声明我们预期有时脚本会执行到一个空的数据库，那么脚本中的DROP语句会失败掉。这个失败的SQL DROP语句会被忽略，但是其他错误会抛出一个异常。这在你的SQL语句不支持DROP … IF EXISTS（或类似语法）时比较有用，但是你想要在重新创建时无条件的移除所有的测试数据。在这个案例中第一个脚本通常是一系列DROP语句的集合，然后是一系列Create语句

ignore-failures选项可以被设置成NONE (默认值), DROPS (忽略失败的删除), or ALL (忽略所有失败).

每个语句都应该以;隔开，或者；字符不存在于所用的脚本的话使用新的一行。你可以全局控制或者单针对一个脚本控制，例如：

```
<jdbc:initialize-database data-source="dataSource" separator="@@
">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql"
separator=";"/>
    <jdbc:script location="classpath:com/foo/sql/db-test-data-1.
sql"/>
    <jdbc:script location="classpath:com/foo/sql/db-test-data-2.
sql"/>
</jdbc:initialize-database>
```

在这个例子中，两个test-data脚本使用@@作为语句分隔符，并且只有db-schema.sql使用;.配置指定默认分隔符是@@，并且将db-schema脚本内容覆盖默认值。

如果你需要从XML命名空间获取更多控制，你可以直接简单的使用DataSourceInitializer，并且在你的应用中将其定义为一个模块

初始化依赖数据库的其他模块

大多数应用只需要使用数据库初始器基本不会遇到其他问题了：这些基本在Spring上下文初始化之前不会使用到数据库。如果你的应用不属于这种情况则需要使用阅读余下的内容。

数据库初始器依赖于Datasource实例，并且在初始化callback方法中执行脚本（类似于XML bean定义中的init-method,组件中的@Postconstruct方法，或是在实现了InitializingBean类的afterPropertiesSet()方法）。如果有其他bean也依赖了同样的数据源同时也在初始化回调模块中使用数据源，那么可能会存在问题因为数据还没有初始化。

一个例子是在应用启动时缓存需要从数据库Load并且初始化数据。

为了解决这个问题你有两个选择：改变你的缓存初始化策略将其移到更后面的阶段，或者确保数据库会首先初始化。

如果你可以控制应用的初始化行为那么第一个选项明显更容易。下面是使用这个方式的一些建议，包括：

- 缓存做懒加载，并且只在第一次访问的时候初始化，这样会提高你应用启动时间。
- 让你的缓存或者其他独立模块通过实现Lifecycle或者SmartLifecycle接口来初始化缓存。当应用上下文启动时如果autoStartup有设置，SmartLifecycle会被自动启动，而Lifecycle 可以在调用ConfigurableApplicationContext.start()时手动启动。
- 使用Spring的ApplicationEvent或者其他类似的自定义监听事件来触发缓存初始化。ContextRefreshedEvent总是在Spring容器加载完毕的时候使用（在所有Bean被初始化之后）。这类事件钩子(hook)机制在很多时候非常有用（这也是SmartLifecycle的默认工作机制）

第二个选项也可以很容易实现。建议如下：

- 依赖Spring Beanfatory的默认行为，Bean会按照注册顺序被初始化。你可以通过在XML配置中设置顺序来指定你应用模块的初始化顺序，确保数据库和数据库初始化会首先被执行。
- 将Datasource和业务模块隔离，通过将它们放在各自独立的ApplicationContext上下文实例来控制其启动顺序。（例如：父上下文包含DataSource，子上下文包含业务模块）。这种结构在Spring Web应用中很常用，同时也是一种通用做法

16. ORM和数据访问

16.1 介绍一下Spring中的ORM

Spring框架在实现资源管理、数据访问对象（DAO）层，和事务策略等方面，支持对Java持久化API（JPA）以及原生Hibernate的集成。以Hibernate举例来说，Spring有非常赞的IoC功能，可以解决许多典型的Hibernate配置和集成问题。开发者可以通过依赖注入来配置O-R（对象关系）映射组件支持的特性。Hibernate的这些特性可以参与Spring的资源和事务管理，并且符合Spring的通用事务和DAO层的异常体系。因此，Spring团队推荐开发者使用Spring集成的方式来开发DAO层，而不是使用原生的Hibernate或者JPA的API。老版本的Spring DAO模板现在不推荐使用了，想了解这部分内容可以参考经典ORM使用一节。

当开发者创建数据访问应用程序时，Spring会为开发者选择的ORM层对应功能进行优化。而且，开发者可以根据需要来利用Spring对集成ORM的支持，开发者应该将此集成工作与维护内部类似的基础架构服务的成本和风险进行权衡。同时，开发者在使用Spring集成的时候可以很大程度上不用考虑技术，将ORM的支持当做一个库来使用，因为所有的组件都被设计为可重用的JavaBean组件了。Spring IoC容器中的ORM十分易于配置和部署。本节中的大多数示例都是讲解在Spring容器中的来如何配置。

开发者使用Spring框架来中创建自己的ORM DAO的好处如下：

- 易于测试。Spring IoC的模式使得开发者可以轻易的替换Hibernate的 `SessionFactory` 实例，JDBC的 `DataSource` 实例，事务管理器，以及映射对象（如果有必要）的配置和实现。这一特点十分利于开发者对每个模块进行独立的测试。
- 泛化数据访问异常。Spring可以将ORM工具的异常封装起来，将所有异常（可以是受检异常）封装成运行时的 `DataAccessException` 体系。这一特性可以令开发者在合适的逻辑层上处理绝大多数不可修复的持久化异常，避免了大量的 `catch` , `throw` 和异常的声明。开发者还可以按需来处理这些异常。其中，JDBC异常（包括一些特定DB语言）都会被封装为相同的体系，意味着开发者即使使用不同的JDBC操作，基于不同的DB，也可以保证一致的编程模型。
- 通用的资源管理。Spring的应用上下文可以通过处理配置源的位置来灵活配置Hibernate的 `SessionFactory` 实例，JPA的 `EntityManagerFactory` 实例，JDBC的 `DataSource` 实例以及其他类似的资源。Spring的这一特性使得这些实例的配置十分易于管理和修改。同时，Spring还为处理持久化资源的配置提

供了高效，易用和安全的处理方式。举个例子，有些代码使用了Hibernate需要使用相同的 `Session` 来确保高效性和正确的事务处理。Spring通过Hibernate的 `SessionFactory` 来获取当前的 `Session` ，来透明的将 `Session` 绑定到当前的线程。Srping为任何本地或者JTA事务环境解决了在使用Hibernate时碰到的一些常见问题。

- 集成事务管理。开发者可以通过 `@Transactional` 注解或在XML配置文件中显式配置事务AOP Advise拦截，将ORM代码封装在声明式的AOP方法拦截器中。事务的语义和异常处理（回滚等）都可以根据开发者自己的需求来定制。在后面的章节中，资源和事务管理中，开发者可以在不影响ORM相关代码的情况下替换使用不同的事务管理器。例如，开发者可以在本地事务和JTA之间进行交换，并在两种情况下具有相同的完整服务（如声明式事务）。而且，JDBC相关的代码在事务上完全和处理ORM部分的代码集成。这对于不适用于ORM的数据访问非常有用，例如批处理和BLOB流式传输，仍然需要与ORM操作共享常见事务。

> 为了更全面的ORM支持，包括支持其他类型的数据库技术（如MongoDB），开发者可能需要查看Spring Data系列项目。如果开发者是JPA用户，则可以从https://spring.io的查阅[开始使用JPA访问数据指南](https://spring.io/guides/gs/accessing-data-jpa/)一文进行简单了解。

16.2 集成ORM的注意事项

本节重点介绍适用于所有集成ORM技术的注意事项。在16.3Hibernate一节中提供了很多关于如何配置和使用这些特性提的信息。

Spring对ORM集成的主要目的是使应用层次化，可以任意选择数据访问和事务管理技术，并且为应用对象提供松耦合结构。不再将业务逻辑依赖于数据访问或者事务策略上，不再使用基于硬编码的资源查找，不再使用难以替代的单例，不再自定义服务的注册。同时，为应用提供一个简单和一致的方法来装载对象，保证他们的重用并且尽可能不依赖于容器。所有单独的数据访问功能都可以自己使用，也可以很好地与Spring的 `ApplicationContext` 集成，提供基于XML的配置和不需要Spring感知的普通 `JavaBean` 实例。在典型的Spring应用程序中，许多重要的对象都是 `JavaBean` ：数据访问模板，数据访问对象，事务管理器，使用数据访问对象和事务管理器的业务服务，Web视图解析器，使用业务服务的Web控制器等等。

16.2.1 资源和事务管理

通常企业应用都会包含很多重复的的资源管理代码。很多项目总是尝试去创造自己的解决方案，有时会为了开发的方便而牺牲对错误的处理。Spring为资源的配置管理提供了简单易用的解决方案，在JDBC上使用模板技术，在ORM上使用AOP拦截技术。

Spring的基础设施提供了合适的资源处理，同时Spring引入了DAO层的异常体系，可以适用于任何数据访问策略。对于JDBC直连来说，前面提及到的 `JdbcTemplate` 类提供了包括连接处理，对 `SQLException` 到 `DataAccessException` 的异常封装，同时还包含对于一些特定数据库SQL错误代码的转换。对于ORM技术来说，可以参考下一节来了解异常封装的优点。

当谈到事务管理时， `JdbcTemplate` 类通过Spring事务管理器挂接到Spring事务支持，并支持JTA和JDBC事务。Spring通过Hibernate，JPA事务管理器和JTA的支持来提供Hibernate和JPA这类ORM技术的支持。想了解更多关于事务的描述，可以参考第13章，事务管理。

16.2.2 异常转义

当在DAO层中使用Hibernate或者JPA的时候，开发者必须决定该如何处理持久化技术的一些原生异常。DAO层会根据选择技术的不同而抛出`HibernateException`或者`PersistenceException`。这些异常都属于运行时异常，所以无需显式声明和捕捉。同时，开发者同时还需要处理`IllegalArgumentException`和`IllegalStateException`这类异常。一般情况下，调用方通常只能将这一类异常视为致命的异常，除非他们想要自己的应用依赖于持久性技术原生的异常体系。如果需要捕获一些特定的错误，比如乐观锁获取失败一类的错误，只能选择调用方和实现策略耦合到一起。对于那些只基于某种特定ORM技术或者不需要特殊异常处理的应用来说，使用ORM本身的异常体系的代价是可以接受的。但是，Spring可以通过`@Repository`注解透明地应用异常转换，以解耦调用方和ORM技术的耦合：

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
```

```
<beans>

    <!-- Exception translation bean post processor -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

上面的后置处理器`PersistenceExceptionTranslationPostProcessor`，会自动查找所有的异常转义器（实现`PersistenceExceptionTranslator`接口的Bean），并且拦截所有标记为`@Repository`注解的Bean，通过代理来拦截异常，然后通过`PersistenceExceptionTranslator`将DAO层异常转义后的异常抛出。

总而言之：开发者可以既基于简单的持久化技术的API和注解来实现DAO，同时还受益于Spring管理的事务，依赖注入和透明异常转换（如果需要）到Spring的自定义异常层次结构。

16.3 Hibernate

我们将首先介绍Spring环境中的Hibernate 5，然后通过使用Hibernate 5来演示Spring集成O-R映射器的方法。本节将详细介绍许多问题，并显示DAO实现和事务划分的不同变体。这些模式中大多数可以直接转换为所有其他支持的ORM工具。本章中的以下部分将通过简单的例子来介绍其他ORM技术。

> 从Spring 5.0开始，Spring需要Hibernate ORM对JPA的支持要基于4.3或更高的版本，甚至Hibernate ORM 5.0+可以针对本机Hibernate Session API进行编程。请注意，Hibernate团队可能不会在5.0之前维护任何版本，仅仅专注于5.2以后的版本。

16.3.1 在Spring容器中配置SessionFactory

开发者可以将资源如JDBC `DataSource` 或Hibernate `SessionFactory` 定义为Spring容器中的bean来避免将应用程序对象绑定到硬编码的资源查找上。应用对象需要访问资源的时候，都通过对应的Bean实例进行间接查找，详情可以通过下一节的DAO的定义来参考。

下面引用的应用的XML元数据定义就展示了如何配置JDBC的 `DataSource` 和 `Hibernate` 的 `SessionFactory` 的：

```
<beans>
    <bean id="myDataSource" class="org.apache.commons.dbcp.Basic
DataSource" destroy-method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcD
river"/>
        <property name="url" value="jdbc:hsqldb:hsql://localhost
:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="mySessionFactory" class="org.springframework.orm.h
ibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.HSQLDial
ect
            </value>
        </property>
    </bean>
</beans>
```

这样，从本地的Jaksrta Commons DBCP的 `BasicDataSource` 转换到JNDI定位的 `DataSource` 仅仅只需要修改配置文件。

```
<beans>
    <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/
jdbc/myds"/>
</beans>
```

开发者也可以通过Spring的 `JndiObjectFactoryBean` 或者 `<jee:jndi-lookup>` 来获取对应Bean以访问JNDI定位的 `SessionFactory` 。但是，JNDI定位的 `SessionFactory` 在EJB上下文不常见。

16.3.2 基于Hibernate API来实现DAO

Hibernate有一个特性称之为上下文会话，在每个Hibernate本身每个事务都管理一个当前的 `Session` 。这大致相当于Spring每个事务的一个Hibernate `Session` 的同步。如下的DAO的实现类就是基于简单的Hibernate API实现的：

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory)
 {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where pr
oduct.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

除了需要在实例中持有 `SessionFactory` 引用以外，上面的代码风格跟Hibernate文档中的例子十分相近。Spring团队强烈建议使用这种基于实例变量的实现风格，而非守旧的 `static HibernateUtil` 风格(总的来说，除非绝对必要，否则尽量不要使用 `static` 变量来持有资源)。

上面DAO的实现完全符合Spring依赖注入的样式：这种方式可以很好的集成Spring IoC容器，就好像Spring的 `HibernateTemplate` 代码一样。当然，DAO层的实现也可以通过纯Java的方式来配置（比如在UT中）。简单实例化 `ProductDaoImpl` 并且调用 `setSessionFactory(...)` 即可。当然，也可以使用Spring bean来进行注入，参考如下XML配置：

```
<beans>
    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>
</beans>
```

上面的DAO实现方式的好处在于只依赖于Hibernate API，而无需引入Spring的class。这从非侵入性的角度来看当然是有吸引力的，毫无疑问，这种开发方式会令Hibernate开发人员将会更加自然。

然而，DAO层会抛出Hibernate自有异常 `HibernateException`（属于非检查异常，无需显式声明和使用try-catch），但是也意味着调用方会将异常看做致命异常——除非调用方将Hibernate异常体系作为应用的异常体系来处理。而在这种情况下，除非调用方自己来实现一定的策略，否则捕获一些诸如乐观锁失败之类的特定错误是不可能的。对于强烈基于Hibernate的应用程序或不需要对特殊异常处理的应用程序，这种代价可能是可以接受的。

幸运的是，Spring的 `LocalSessionFactoryBean` 可以通过Hibernate的 `SessionFactory.getCurrentSession()` 方法为所有的Spring事务策略提供支持，使用 `HibernateTransactionManager` 返回当前的Spring管理的事务的 `Session` 。当然，该方法的标准行为仍然是返回与正在进行的JTA事务相关联的当前 `Session` （如果有的话）。无论开发者是使用Spring的 `JtaTransactionManager` ，EJB容器管理事务（CMT）还是JTA，都会适用此行为。

总而言之：开发者可以基于纯Hibernate API来实现DAO，同时也可以集成Spring来管理事务。

16.3.3 声明式事务划分

Spring团队建议开发者使用Spring声明式的事务支持，这样可以通过AOP事务拦截器来替代事务API的显式调用。AOP事务拦截器可以在Spring容器中使用XML或者Java的注解来进行配置。这种事务拦截器可以令开发者的代码和重复的事务代码相解耦，而开发者可以将精力更多集中在业务逻辑上，而业务逻辑才是应用的核心。

> 在继续之前，强烈建议开发者先查阅章节13.5 声明式事务管理的内容。

开发者可以在服务层的代码使用注解 `@Transactional` ，这样可以让Spring容器找到这些注解，以对其中注解了的方法提供事务语义。

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Transactional
    public void increasePriceOfAllProductsInCategory(final Strin
g category) {
        List productsToChange = this.productDao.loadProductsByCa
tegory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }

}
```

开发者所需要做的就是在容器中配置 `PlatformTransactionManager` 的实现，或者是在XML中配置 `<tx:annotation-driver/>` 标签，这样就可以在运行时支持 `@Transactional` 的处理了。参考如下XML代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
">

    <!-- SessionFactory, DataSource, etc. omitted -->

    <bean id="transactionManager"
            class="org.springframework.orm.hibernate5.HibernateT
ransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

    <bean id="myProductService" class="product.SimpleProductServ
ice">
        <property name="productDao" ref="myProductDao"/>
    </bean>
</beans>
```

16.3.4 编程式事务划分

开发者可以在应用程序的更高级别上对事务进行标定，而不用考虑低级别的数据访问执行了多少操作。这样不会对业务服务的实现进行限制；只需要定义一个Spring的 `PlatformTransactionManager` 即可。当然， `PlatformTransactionManager` 可以从多处获取，但最好是通

过 `setTransactionManager(..)` 方法以Bean来注入，正如 `ProductDAO` 应该由 `setProductDao(..)` 方法配置一样。下面的代码显示Spring应用程序上下文中的事务管理器和业务服务的定义，以及业务方法实现的示例：

```
<bean id="myTxManager" class="org.springframework.orm.hibernate5
.HibernateTransactionManager">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImp
l">
        <property name="transactionManager" ref="myTxManager"/>
        <property name="productDao" ref="myProductDao"/>
    </bean>
</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager
 transactionManager) {
        this.transactionTemplate = new TransactionTemplate(trans
actionManager);
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final Strin
g category) {
        this.transactionTemplate.execute(new TransactionCallback
WithoutResult() {
            public void doInTransactionWithoutResult(Transaction
Status status) {
                List productsToChange = this.productDao.loadProd
uctsByCategory(category);
                // do the price increase...
            }
        });
    }
}
```

Spring的 `TransactionInterceptor` 允许任何检查的应用异常到 `callback` 代码中去，而 `TransactionTemplate` 还会非受检异常触发进行回调。 `TransactionTemplate` 则会因为非受检异常或者是由应用标记事务回滚(通过 `TransactionStatus`)。 `TransactionInterceptor` 也是一样的处理逻辑，但是同时还允许基于方法配置回滚策略。

16.3.5 事务管理策略

无论是 `TransactionTemplate` 或者是 `TransactionInterceptor` 都将实际的事务处理代理到 `PlatformTransactionManager` 实例上来进行处理的，这个实例的实现可以是一个 `HibernateTransactionManager` (包含一个Hibernate的 `SessionFactory` 通过使用 `ThreadLocal` 的 `Session`)，也可以是 `JatTransactionManager` (代理到容器的JTA子系统)。开发者甚至可以使用一个自定义的 `PlatformTransactionManager` 的实现。现在，如果应用有需求需要需要部署分布式事务的话，只是一个配置变化，就可以从本地Hibernate事务管理切换到JTA。简单地用Spring的JTA事务实现来替换Hibernate事务管理器即可。因为引用的 `PlatformTransactionManager` 的是通用事务管理API，事务管理器之间的切换是无需修改代码的。

对于那些跨越了多个Hibernate会话工厂的分布式事务，只需要将 `JtaTransactionManager` 和多个 `LocalSessionFactoryBean` 定义相结合即可。每个DAO之后会获取一个特定的 `SessionFactory` 引用。如果所有底层JDBC数据源都是事务性容器，那么只要使用 `JtaTransactionManager` 作为策略实现，业务服务就可以划分任意数量的DAO和任意数量的会话工厂的事务。

无论是 `HibernateTransactionManager` 还是 `JtaTransactionManager` 都允许使用JVM级别的缓存来处理Hibernate，无需基于容器的事务管理器查找，或者JCA连接器（如果开发者没有使用EJB来实例化事务的话）。

`HibernateTransactionManager` 可以为指定的数据源的Hibernate JDBC的 `Connection` 转成为纯JDBC的访问代码。如果开发者仅访问一个数据库，则开发者完全可以不使用JTA，通过Hibernate和JDBC数据访问进行高级别事务划分。如果开发者已经通过 `LocalSessionFactoryBean` 的 `dataSource` 属性与 `DataSource` 设置了传入的 `SessionFactory` ， `HibernateTransactionManager` 会自动将Hibernate事务公开为JDBC事务。或者，开发者可以通过 `HibernateTransactionManager` 的 `dataSource` 属性的配置以确定公开事务的类型。

16.3.6 对比由容器管理的和本地定义的资源

开发者可以在不修改一行代码的情况下，在容器管理的JNDI `SessionFactory` 和本地定义的 `SessionFactory` 之间进行切换。是否将资源定义保留在容器中，还是仅仅留在应用中，都取决于开发者使用的事务策略。相对于Spring定义的本

地 `SessionFactory` 来说，手动注册的JNDI `SessionFactory` 没有什么优势。通过Hibernate的JCA连接器来发布一个 `SessionFactory` 只会令代码更符合J2EE服务标准，但是并不会带来任何实际的价值。

Spring对事务支持不限于容器。使用除JTA之外的任何策略配置，事务都可以在独立或测试环境中工作。特别是在单数据库事务的典型情况下，Spring的单一资源本地事务支持是一种轻量级和强大的替代JTA的解决方案。当开发者使用本地EJB无状态会话Bean来驱动事务时，即使只访问单个数据库，并且只使用无状态会话Bean来通过容器管理的事务来提供声明式事务，开发者的代码依然是依赖于EJB容器和JTA的。同时，以编程方式直接使用JTA也需要一个J2EE环境的。 JTA不涉及JTA本身和JNDI DataSource实例方面的容器依赖关系。对于非Spring，JTA驱动的Hibernate事务，开发者必须使用Hibernate JCA连接器或开发额外的Hibernate事务代码，并为JVM级缓存正确配置 `TransactionManagerLookup` 。

Spring驱动的事务可以与本地定义的Hibernate `SessionFactory` 一样工作，就像本地JDBC DataSource访问单个数据库一样。但是，当开发者有分布式事务的要求的情况下，只能选择使用Spring JTA事务策略。JCA连接器是需要特定容器遵循一致的部署步骤的，而且显然JCA支持是需要放在第一位的。JCA的配置需要比部署本地资源定义和Spring驱动事务的简单web应用程序需要更多额外的的工作。同时，开发者还需要使用容器的企业版，比如，如果开发者使用的是WebLogic Express的非企业版，就是不支持JCA的。具有跨越单个数据库的本地资源和事务的Spring应用程序适用于任何基于J2EE的Web容器（不包括JTA，JCA或EJB），如Tomcat，Resin或甚至是Jetty。此外，开发者可以轻松地在桌面应用程序或测试套件中重用中间层代码。

综合前面的叙述，如果不使用EJB，请尽量使用本地的 `SessionFactory` 设置和Spring的 `HibernateTransactionManager` 或 `JtaTransactionManager` 。开发者能够得到了前面提到的所有好处，包括适当的事务性JVM级缓存和分布式事务支持，而且没有容器部署的不便。只有必须配合EJB使用的时候，JNDI通过JCA连接器来注册Hibernate `SessionFactory` 才有价值。

16.3.7 Hibernate的虚假应用服务器警告

在某些具有非常严格的 `XADataSource` 实现的JTA环境（目前只有一些WebLogic Server和WebSphere版本）中，当配置Hibernate时，没有考虑到JTA的 `PlatformTransactionManager` 对象，可能会在应用程序服务器日志中显示虚假

警告或异常。这些警告或异常经常描述正在访问的连接不再有效，或者JDBC访问不再有效。这通常可能是因为事务不再有效。例如，这是WebLogic的一个实际异常：

```
java.sql.SQLException: The transaction is no longer active - sta
tus: 'Committed'. No
further JDBC access is allowed within this transaction.
```

开发者可以通过配置令Hibernate意识到Spring中同步的JTA `PlatformTransactionManager` 实例的存在，这样即可消除掉前面所说的虚假警告信息。开发者有以下两种选择：

- 如果在应用程序上下文中，开发者已经直接获取了JTA `PlatformTransactionManager` 对象（可能是从JNDI到 `JndiObjectFactoryBean` 或者 `<jee:jndi-lookup>` 标签），并将其提供给Spring的 `JtaTransactionManager` （其中最简单的方法就是指定一个引用bean将此JTA `PlatformTransactionManager` 实例定义为 `LocalSessionFactoryBean` 的 `jtaTransactionManager` 属性的值）。Spring之后会令 `PlatformTransactionManager` 对象对Hibernate可见。
- 更有可能开发者无法获取JTA `PlatformTransactionManager` 实例，因为Spring的 `JtaTransactionManager` 是可以自己找到该实例的。因此，开发者需要配置Hibernate令其直接查找JTA `PlatformTransactionManager` 。开发者可以如Hibernate手册中所述那样通过在Hibernate配置中配置应用程序服务器特定的 `TransactionManagerLookup` 类来执行此操作。

本节的其余部分描述了在 `PlatformTransactionManager` 对Hibernate可见和 `PlatformTransactionManager` 对Hibernate不可见的情况下发生的事件序列:

当Hibernate未配置任何对JTA `PlatformTransactionManager` 的进行查找时，JTA事务提交时会发生以下事件：

- JTA事务提交
- Spring的 `JtaTransactionManager` 与JTA事务同步，所以它被JTA事务管理器通过 `afterCompletion` 回调调用。
- 在其他活动中，此同步令Spring通过Hibernate的 `afterTransactionCompletion` 触发回调（用于清除Hibernate缓存），然后在Hibernate Session上调用 `close()` ，从而令Hibernate尝试 `close()` JDBC连接。

- 在某些环境中，因为事务已经提交，应用程序服务器会认为 `Connection` 不可用，导致 `Connection.close()` 调用会触发警告或错误。

当Hibernate配置了对JTA `PlatformTransactionManager` 进行查找时，JTA事务提交时会发生以下事件：

- JTA事务准备提交
- Spring的 `JtaTransactionManager` 与JTA事务同步，所以JTA事务管理器通过 `beforeCompletion` 方法来回调事务。
- Spring确定Hibernate与JTA事务同步，并且行为与前一种情况不同。假设Hibernate Session需要关闭，Spring将会关闭它。
- JTA事务提交。
- Hibernate与JTA事务同步，所以JTA事务管理器通过 `afterCompletion` 方法回调事务，可以正确清除其缓存。

16.4 JPA

Spring JPA在 `org.springframework.orm.jpa` 包中已经可用，Spring JPA用了Hibernate集成相似的方法来提供更易于理解的JPA支持，与此同时，了解了JPA底层实现，可以理解更多的Spring JPA特性。

16.4.1 Spring中JPA配置的三个选项

Spring JPA支持提供了三种配置JPA `EntityManagerFactory` 的方法，之后通过 `EntityManagerFactory` 来获取对应的实体管理器。

LocalEntityManagerFactoryBean

> 通常只有在简单的部署环境中使用此选项，例如在独立应用程序或者进行集成测试时，才会使用这种方式。

`LocalEntityManagerFactoryBean` 创建一个适用于应用程序且仅使用JPA进行数据访问的简单部署环境的 `EntityManagerFactory` 。工厂bean会使用JPA `PersistenceProvider` 自动检测机制，并且在大多数情况下，仅要求开发者指定持久化单元的名称：

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEnt
ityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenc
eUnit"/>
    </bean>
</beans>
```

这种形式的JPA部署是最简单的，同时限制也很多。开发者不能引用现有的JDBC `DataSource` bean定义，并且不支持全局事务。而且，持久化类的织入(weaving)(字节码转换)是特定于提供者的，通常需要在启动时指定特定的JVM代理。该选项仅适用于符合JPA Spec的独立应用程序或测试环境。

从JNDI中获取EntityManagerFactory

> 在部署到J2EE服务器时可以使用此选项。检查服务器的文档来了解如何将自定义JPA提供程序部署到服务器中，从而对服务器进行比默认更多的个性化定制。

从JNDI获取 `EntityManagerFactory` (例如在Java EE环境中)，只需要在XML配置中加入配置信息即可：

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersist
enceUnit"/>
</beans>
```

此操作将采用标准J2EE引导：J2EE服务器自动检测J2EE部署描述符（例如web.xml）中persistence-unit-ref条目和持久性单元（实际上是应用程序jar中的META-INF/persistence.xml文件），并为这些持久性单元定义环境上下文位置。

在这种情况下，整个持久化单元部署（包括持久化类的织入(weaving)（字节码转换））都取决于J2EE服务器。JDBC `DataSource` 通过META-INF/persistence.xml文件中的JNDI位置进行定义; 而 `EntityManager` 事务与服务器JTA子系统集成。 Spring仅使用获取的 `EntityManagerFactory` ，通过依赖注入将其传递给应用程序对象，通常通过 `JtaTransactionManager` 来管理持久性单元的事务。

如果在同一应用程序中使用多个持久性单元，则这种JNDI检索的持久性单元的bean名称应与应用程序用于引用它们的持久性单元名称相匹配，例如 `@PersistenceUnit` 和 `@PersistenceContext` 注释。

LocalContainerEntityManagerFactoryBean

> 在基于Spring的应用程序环境中使用此选项来实现完整的JPA功能。这包括诸如Tomcat的Web容器，以及具有复杂持久性要求的独立应用程序和集成测试。

`LocalContainerEntityManagerFactoryBean` 可以完全控制 `EntityManagerFactory` 的配置，同时适用于需要细粒度定制的环境。 `LocalContainerEntityManagerFactoryBean` 会基于 `persistence.xml` 文件， `dataSourceLookup` 策略和指定的 `loadTimeWeaver` 来创建一

个 `PersistenceUnitInfo` 实例。因此，可以在JNDI之外使用自定义数据源并控制织入(weaving)过程。以下示例显示 `LocalContainerEntityManagerFactoryBean` 的典型Bean定义：

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="someDataSource"/>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
    </bean>
</beans>
```

下面的例子是一个典型的persistence.xml文件：

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
        <mapping-file>META-INF/orm.xml</mapping-file>
        <exclude-unlisted-classes/>
    </persistence-unit>
</persistence>
```

> `<exclude-unlisted-classes />` 标签表示不会进行注解实体类的扫描。指定的显式 `true` 值 - `<exclude-unlisted-classes>true</exclude-unlisted-classes/>` - 也意味着不进行扫描。 `<exclude-unlisted-classes> false</exclude-unlisted-classes>` 则会触发扫描;但是，如果开发者需要进行实体类扫描，建议开发者简单地省略 `<exclude-unlisted-classes>` 元素。

`LocalContainerEntityManagerFactoryBean` 是最强大的JPA设置选项，允许在应用程序中进行灵活的本地配置。它支持连接到现有的JDBC `DataSource` ，支持本地和全局事务等。但是，它对运行时环境施加了需求，其中之一就是如果持久性提供程序需要字节码转换，就需要有织入(weaving)能力的类加载器。

此选项可能与J2EE服务器的内置JPA功能冲突。在完整的J2EE环境中，请考虑从JNDI获取 `EntityManagerFactory` 。或者，在开发者的 `LocalContainerEntityManagerFactoryBean` 定义中指定一个自定义 `persistenceXmlLocation` ，例如META-INF/my-persistence.xml，并且只在应用程序jar文件中包含有该名称的描述符。因为J2EE服务器仅查找默认的META-INF/persistence.xml文件，所以它会忽略这种自定义持久性单元，从而避免了与Spring驱动的JPA设置之间发生冲突。（例如，这适用于Resin 3.1）

> 何时需要加载时间织入？
> 并非所有JPA提供商都需要JVM代理。Hibernate就是一个不需要JVM代理的例子。如果开发者的提供商不需要代理或开发者有其他替代方案，例如通过定制编译器或 `Ant` 任务在构建时应用增强功能，则不用使用加载时间编织器。

`LoadTimeWeaver` 是一个Spring提供的接口，它允许以特定方式插入JPA `ClassTransformer` 实例，这取决于环境是Web容器还是应用程序服务器。通过代理挂载 `ClassTransformers` 通常性能较差。代理会对整个虚拟机进行操作，并检查加载的每个类，这是生产服务器环境中最不需要的额外负载。

Spring为各种环境提供了一些 `LoadTimeWeaver` 实现，允许 `ClassTransformer` 实例仅适用于每个类加载器，而不是每个VM。

有关 `LoadTimeWeaver` 的实现及其设置的通用或定制的各种平台（如Tomcat，WebLogic，GlassFish，Resin和JBoss）的更多了解，请参阅AOP章节中的Spring配置一节。

如前面部分所述，开发者可以使用 `@EnableLoadTimeWeaving` 注解或者 `load-time-weaver` XML元素来配置上下文范围的 `LoadTimeWeaver` 。所有JPA `LocalContainerEntityManagerFactoryBeans` 都会自动拾取这样的全局织入器。这是设置加载时间织入器的首选方式，为平台（WebLogic，GlassFish，Tomcat，Resin，JBoss或VM代理）提供自动检测功能，并将织入组件自动传播到所有可以感知织入者的Bean：

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainer
EntityManagerFactoryBean">
    ...
</bean>
```

开发者也可以通过 `LocalContainerEntityManagerFactoryBean` 的 `loadTimeWeaver` 属性来手动指定专用的织入器：

```
<bean id="emf" class="org.springframework.orm.jpa.LocalContainer
EntityManagerFactoryBean">
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading
.ReflectiveLoadTimeWeaver"/>
    </property>
</bean>
```

无论LTW如何配置，使用这种技术，依赖于仪器的JPA应用程序都可以在目标平台（例如：Tomcat）中运行，而不需要代理。这尤其重要的是当主机应用程序依赖于不同的JPA实现时，因为JPA转换器仅应用于类加载器级，彼此隔离。

处理多个持久化单元

例如，对于依赖存储在类路径中的各种JARS中的多个持久性单元位置的应用程序，Spring将 `PersistenceUnitManager` 作为中央仓库来避免可能昂贵的持久性单元发现过程。默认实现允许指定多个位置，这些位置将通过持久性单元名称进行解析并稍后检索。（默认情况下，搜索classpath下的META-INF/persistence.xml文件。）

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceuni
t.DefaultPersistenceUnitManager">
    <property name="persistenceXmlLocations">
        <list>
            <value>org/springframework/orm/jpa/domain/persistenc
e-multi.xml</value>
            <value>classpath:/my/package/**/custom-persistence.x
ml</value>
            <value>classpath*:META-INF/persistence.xml</value>
        </list>
    </property>
    <property name="dataSources">
        <map>
            <entry key="localDataSource" value-ref="local-db"/>
            <entry key="remoteDataSource" value-ref="remote-db"/
>
        </map>
    </property>
    <!-- if no datasource is specified, use this one -->
    <property name="defaultDataSource" ref="remoteDataSource"/>
</bean>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainer
EntityManagerFactoryBean">
    <property name="persistenceUnitManager" ref="pum"/>
    <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

在默认实现传递给JPA provider之前，是允许通过属性（影响全部持久化单元）或者通过 `PersistenceUnitPostProcessor` 以编程(对选择的持久化单元进行)进行对 `PersistenceUnitInfo` 进行自定义的。如果没有指定 `PersistenceUnitManager` ，则由 `LocalContainerEntityManagerFactoryBean` 在内部创建和使用。

16.4.2 基于JPA的EntityManagerFactory和EntityManager来实现DAO

> 虽然 `EntityManagerFactory` 实例是线程安全的，但 `EntityManager` 实例不是。注入的JPA `EntityManager` 的行为类似于从JPA Spec中定义的应用程序服务器的JNDI环境中提取的 `EntityManager` 。它将所有调用委托给当前事务的 `EntityManager` (如果有);否则，它每个操作返回的都是新创建的 `EntityManager` ，通过使用不同的 `EntityManager` 来保证使用时的线程安全。

通过注入的方式使用 `EntityManagerFactory` 或 `EntityManager` 来编写JPA代码，是不需要依赖任何Spring定义的类的。如果启用了 `PersistenceAnnotationBeanPostProcessor` ，Spring可以在实例级别和方法级别识别 `@PersistenceUnit` 和 `@PersistenceContext` 注解。使用 `@PersistenceUnit` 注解的纯JPA DAO实现可能如下所示：

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf
) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
            Query query = em.createQuery("from Product as p wher
e p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

上面的DAO对Spring的实现是没有任何依赖的，而且很适合与Spring的应用程序上下文进行集成。而且，DAO还可以通过注解来注入默认的 `EntityManagerFactory` ：

```
<beans>

    <!-- bean post-processor for JPA annotations -->
    <bean class="org.springframework.orm.jpa.support.Persistence
AnnotationBeanPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>
</beans>
```

如果不想明确定义 `PersistenceAnnotationBeanPostProcessor` ，可以考虑在应用程序上下文配置中使用Spring上下文 `annotation-config` XML元素。这样做会自动注册所有Spring标准后置处理器，用于初始化基于注解的配置，包括 `CommonAnnotationBeanPostProcessor` 等。

```
<beans>
    <!-- post-processors for all standard config annotations -->
    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>
</beans>
```

这样的DAO的主要问题是它总是通过工厂创建一个新的 `EntityManager` 。开发者可以通过请求事务性 `EntityManager` （也称为共享*EntityManager*，因为它是实际的事务性EntityManager的一个共享的，线程安全的代理）来避免这种情况。

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery("from Product as p where p.
category = :category");
        query.setParameter("category", category);
        return query.getResultList();
    }
}
```

`@PersistenceContext` 注解具有可选的属性类型，默认值为 `PersistenceContextType.TRANSACTION` 。此默认值是开发者所需要接收共享的 `EntityManager` 代理。替代方案 `PersistenceContextType.EXTENDED` 则完全不同：该方案会返回一个所谓扩展的 `EntityManager` ，该 `EntityManager` 不是线程安全的，因此不能在并发访问的组件（如Spring管理的单例Bean）中使用。扩展实体管理器仅应用于状态组件中，比如持有会话的组件，其中 `EntityManager` 的生命周期与当前事务无关，而是完全取决于应用程序。

> 方法和实例变量级别注入
> 指示依赖注入（例如 `@PersistenceUnit` 和 `@PersistenceContext` ）的注解可以应用于类中的实例变量或方法，也就是表达式方法级注入和实例变量级注入。实例变量级注释简洁易用，而方法级别允许进一步处理注入的依赖关系。在这两种情况下，成员的可见性（ `public` ， `protected` ， `private` ）并不重要。
> 类级注解怎么办？
> 在J2EE平台上，它们用于依赖关系声明，而不是资源注入。

注入的 `EntityManager` 是由Spring管理的（Spring可以意识到正在进行的事务）。重要的是要注意，因为通过注解进行注入，即使新的DAO实现使用通过方法注入的 `EntityManager` 而不是 `EntityManagerFactory` 的注入的，在应用程序上下文XML中不需要进行任何修改。

这种DAO风格的主要优点是它只依赖于Java Persistence API;不需要导入任何Spring的实现类。而且，Spring容器可以识别JPA注解来实现自动的注入和管理。从非侵入的角度来看，这种风格对JPA开发者来说可能更为自然。

16.4.3 Spring驱动的JPA事务

> 如果开发者还没有阅读声明式事务管理，强烈建议开发者先行阅读，这样可以更详细地了解Spring的对声明式事务支持。

JPA的推荐策略是通过JPA的本地事务支持的本地事务。 Spring的 `JpaTransactionManager` 提供了许多来自本地JDBC事务的功能，例如针对任何常规JDBC连接池（不需要XA要求）指定事务的隔离级别和资源级只读优化等。

Spring JPA还允许配置 `JpaTransactionManager` 将JPA事务暴露给访问同一 `DataSource` 的JDBC访问代码，前提是注册的 `JpaDialect` 支持检索底层JDBC连接。Spring为EclipseLink和Hibernate JPA实现提供了实现。有关 `JpaDialect` 机制的详细信息，请参阅下一节。

16.4.4 JpaDialect和JpaVendorAdapter

作为高级功能， `JpaTransactionManager` 和 `AbstractEntityManagerFactoryBean` 的子类支持自定义 `JpaDialect` ，将其作为Bean传递给 `jpaDialect` 属性。 `JpaDialect` 实现可以以供应商特定的方式使能Spring支持的一些高级功能：

- 应用特定的事务语义，如自定义隔离级别或事务超时
- 为基于JDBC的DAO导出事务性JDBC连接
- 从 `PersistenceExceptions` 到Spring `DataAccessExceptions` 的异常转义

这对于特殊的事务语义和异常的高级翻译特别有价值。但是Spring使用的默认实现（ `DefaultJpaDialect` ）是不提供任何特殊功能的。如果需要上述功能，则必须指定适当的方言才可以。

> 作为一个更广泛的供应商适应设施，主要用于Spring的全功能 `LocalContainerEntityManagerFactoryBean` 设置， `JpaVendorAdapter` 将 `JpaDialect` 的功能与其他提供者特定的默认设置相结合。指定 `HibernateJpaVendorAdapter` 或 `EclipseLinkJpaVendorAdapter` 是分别为Hibernate或EclipseLink自动配置 `EntityManagerFactory` 设置的最简单方便的方法。但是请注意，这些提供程序适配器主要是为了与Spring驱动的事务管理一起使用而设计的，即为了与 `JpaTransactionManager` 配合使用的。

有关其操作的更多详细信息以及在Spring的JPA支持中如何使用，请参阅 `JpaDialect` 和 `JpaVendorAdapter` 的Javadoc。

16.4.5 为JPA配置JTA事务管理

作为 `JpaTransactionManager` 的替代方案，Spring还允许通过JTA在J2EE环境中或与独立的事务协调器（如Atomikos）进行多资源事务协调。除了用Spring的 `JtaTransactionManager` 替换 `JpaTransactionManager` ，还有需要以下一些操作：

- 底层JDBC连接池是需要具备XA功能，并与开发者的事务协调器集成的。这在J2EE环境中很简单，只需通过JNDI导出不同类型的 `DataSource` 即可。有关导出 `DataSource` 等详细信息，可以参考应用服务器文档。类似地，独立的事务协调器通常带有特殊的XA集成的 `DataSource` 实现。
- 需要为JTA配置JPA `EntityManagerFactory` 。这是特定于提供程序的，通常通过在 `LocalContainerEntityManagerFactoryBean` 的特殊属性指定为"jpaProperties"。在使用Hibernate的情况下，这些属性甚至是需要基于特定的版本的;请查阅Hibernate文档以获取详细信息。
- Spring的 `HibernateJpaVendorAdapter` 会强制执行某些面向Spring的默认设置，例如在Hibernate 5.0中匹配Hibernate自己的默认值的连接释放模式“on-close”，但在5.1 / 5.2中不再存在。对于JTA设置，不要声明 `HibernateJpaVendorAdapter` 开始，或关闭其 `prepareConnection` 标志。或者，将Hibernate 5.2的 `hibernate.connection.handling_mode` 属性设置为 `DELAYED_ACQUISITION_AND_RELEASE_AFTER_STATEMENT` 以恢复Hibernate自己的默认值。有关WebLogic的相关说明，请参考Hibernate的虚假应用服务器警告一节。
- 或者，可以考虑从应用程序服务器本身获取 `EntityManagerFactory` ，即通过JNDI查找而不是本地声明的 `LocalContainerEntityManagerFactoryBean` 。服务器提供的 `EntityManagerFactory` 可能需要在服务器配置中进行特殊定义，减少了部署的移植性，但是 `EntityManagerFactory` 将为开箱即用的服务器JTA环境设置。

17. 使用 O/X(Object/XML)映射器对XML进行编组

17.1 简介

本章将讨论Spring对于 对象/XML 映射的支持。对象/XML 映射，或 O/X 映射，是指将 XML 文档与 XML 文档对象进行互相转换的操作。这一转换操作也被称作 XML 编组，或 XML 序列化。在本章中，这几个概念都指的是同一个东西。 在 O/X 映射中，将一组对象序列化为 XML 的操作是由一个编组器负责的。与之相对，一个反编组器则被用于将 XML 反序列化为一组对象。而这些操作中的 XML 文件来源可能是一份 DOM 文档，一个输入/输出流，或一个 SAX 管理器。 使用 Spring 提供的支持来实现你的 O/X 映射需求具有如下一些好处：

17.1.1 便于配置

Spring 的 bean 工厂使得无需构建 JAXB 上下文、JiBX 绑定工厂就可以配置编组器。你可以像配置你的应用中任何一个 Spring bean 一样地配置编组器。另外，相当一部分编组器可以使用基于 XML 架构的配置来进行设置，这让配置工作变得更为容易。

17.1.2 一致的接口

Spring 的 O/X 映射通过两个全局的接口来执行操作：Marshaller 和 Unmarshaller。这一结构让用户可以在几乎不需要修改编组操作类的前提下，轻易地在不同的 O/X 映射框架之间进行切换。这一结构的另一优势是可以以一种非侵入的方式在代码中混合多种 XML 编组方法（比如有一些编组实现使用 JAXB，而另一些则使用 Castor），从而将各种技术的优势在应用中加以综合利用。

17.1.3 一致的异常继承

Spring 对来自底层 O/X 映射工具的异常进行了转换，以 XmlMappingException 的形式使之成为 Spring 自身异常继承体系的一部分。这些 Spring 运行时异常将初始异常封装其中，因此所有异常信息都会被完整地保留下来。

17.2 编组器与反编组器

就如在“简介”中提到的，一个编组器负责将一个对象序列化成 XML，而一个反编组器则将 XML 流反序列化为一个对象。我们将在本节对 Spring 提供的两个相关接口进行描述。

17.2.1 编组器

Spring 将所有编组操作抽象成了 org.springframework.oxm.Marshaller 中的方法，以下是该接口最主要的一个方法：

```
public interface Marshaller {
  /**
    * 将对象编组并存放在 Result 中.
    */
  void marshal(Object graph, Result result) throws XmlMappingExc
eption, IOException;
}
```

Marshaller 接口有一个主方法用于将一个给定对象编组为一个给定的 javax.xml.transform.Result 实现。这里的 Result 是一个用于代表某种 XML 输出格式的标记接口：不同的 Result 实现会封装不同的 XML 表现形式，详见下表：

Result 的实现	**封装的 XML 表现形式**
DOMResult	org.w3c.dom.Node
SAXResult	org.xml.sax.ContentHandler
StreamResult	java.io.File, java.io.OutputStream, or java.io.Writer

尽管 marshal() 方法的第一个参数只是一个简单对象，但大多数 Marshaller 实现并不真的能处理任意类型的对象。要么这个对象必须在映射文件中定义过映射关系，要么被注解所修饰，要么在编组器中进行过注册，要么与编组器实现拥有共同的基类。参考后面的章节来确定你所选用的 O/X 技术实现具体是怎么做的。

17.2.2 反编组器

与 Marshaller 接口相对应，还有一个 org.springframework.oxm.Unmarshaller 接口。

```
public interface Unmarshaller {
    /**
      *  将来源 XML 反编组成一个对象
      */
    Object unmarshal(Source source) throws XmlMappingException, I
OException;
}
```

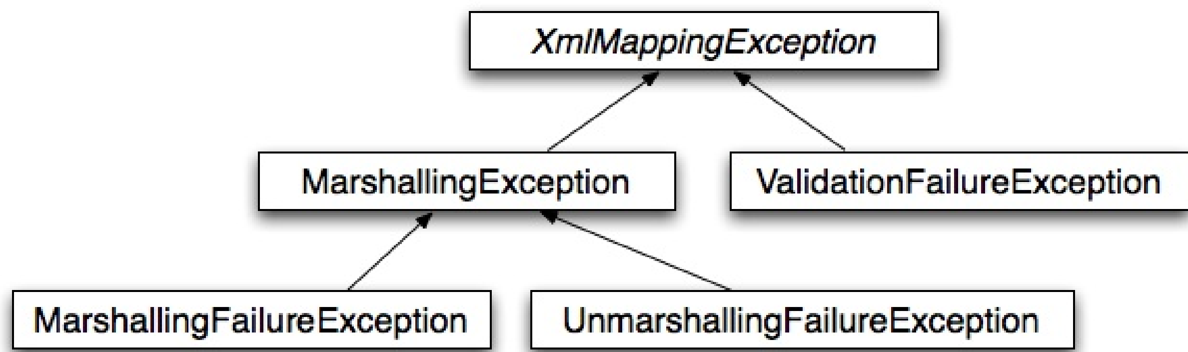
此接口同样也有一个方法，从 javax.xml.transform.Source （一个抽象的 XML 输入）读取 XML 数据，并返回一个相对应的 Java 对象。和 Result 接口一样，Source 是一个拥有三个具体实现的标记接口。每一个实现封装了一种 XML 表现形式。详见下表：

Source 的实现	**封装的 XML 表现形式**
DOMSource	org.w3c.dom.Node
SAXSource	org.xml.sax.InputSource, and org.xml.sax.XMLReader
StreamSource	java.io.File, java.io.OutputStream, or java.io.Writer

17.2.3 XmlMappingException

Spring 对来自底层 O/X 映射工具的异常进行了转换，以 XmlMappingException 的形式使之成为 Spring 自身异常继承体系的一部分。 这些 Spring 运行时异常将初始异常封装其中，因此所有异常信息都会被完整地保留下来。 额外地，虽然底层的 O/X 映射工具并未提供支持，但 MarshallingFailureException 和 UnmarshallingFailureException 让编组与反编组操作中产生的异常得以能够被区分开来。 The O/X Mapping exception hierarchy is shown in the following figure:

以下是 O/X 映射异常的继承层次：


XmlMappingException
MarshallingException
ValidationFailureException
MarshallingFailureException
UnmarshallingFailureException

17.3 Marshaller 与 Unmarshaller 的使用

Spring 的 OXM 可被用于十分广泛的场景。在以下的例子中，我们将使用这一功能将一个由 Spring 管理的应用程序的配置编组为一个 XML 文件。我们用了一个简单的 JavaBean 来表示这些配置：

```
public class Settings {
    private boolean fooEnabled;
    public boolean isFooEnabled() {
        return fooEnabled;
    }
    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

应用程序的主类使用这个 bean 来存放应用的配置信息。除了主要方法外，主类还包含下面两个方法：saveSettings() 将配置 bean 保存成一个名为 settings.xml 的文件， loadSettings() 则将配置信息从 XML 文件中读取出来。另有一个 main() 方法负责构建 Spring 应用上下文，并调用前述两个方法。

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicati
onContext;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;
public class Application {
    private static final String FILE_NAME = "settings.xml";
    private Settings settings = new Settings();
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;
```

```
    public void setMarshaller(Marshaller marshaller) {
        this.marshaller = marshaller;
    }

    public void setUnmarshaller(Unmarshaller unmarshaller) {
        this.unmarshaller = unmarshaller;
    }

    public void saveSettings() throws IOException {
        FileOutputStream os = null;
        try {
            os = new FileOutputStream(FILE_NAME);
            this.marshaller.marshal(settings, new StreamResult(o
s));
        } finally {
            if (os != null) {
                os.close();
            }
        }
    }

    public void loadSettings() throws IOException {
        FileInputStream is = null;
        try {
            is = new FileInputStream(FILE_NAME);
            this.settings = (Settings) this.unmarshaller.unmarsh
al(new StreamSource(is));
        } finally {
            if (is != null) {
                is.close();
            }
        }
    }

    public static void main(String[] args) throws IOException {
        ApplicationContext appContext =
                new ClassPathXmlApplicationContext("applicationC
ontext.xml");
        Application application = (Application) appContext.getBe
an("application");
```

```
            application.saveSettings();
            application.loadSettings();
        }
}
```

需要将 marshaller 和 unmarshaller 这两个属性赋值才能使 Application 正确运行。我们可以使用以下 applicationContext.xml 的内容来实现这一目的：

```
<beans>
    <bean id="application" class="Application">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>
    <bean id="castorMarshaller" class="org.springframework.oxm.c
astor.CastorMarshaller"/>
</beans>
```

该应用中使用了 Castor 这一编组器实例，但我们可以使用在本章稍后描述的任何一个编组器实例来替换 Castor。Castor 默认并不需要任何进一步的配置，所以 bean 定义十分简洁。另外由于 CastorMarshaller 同时实现了 Marshaller 与 Unmarshaller 接口，所以我们可以同时把 castorMarshaller bean 赋值给应用的 marshaller 与 unmarshaller 属性。

此范例应用将会产生如下 settings.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

17.4 基于 XML 架构的配置

可以使用来自 OXM 命名空间的 XML 标签是对编组器的配置变得更简洁。要使用这些标签，请在 XML 文件开头引用恰当的 XML 架构。以下是一个引用 oxm 的示例，请注意粗体字部分：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oxm="http://www.springframework.org/schema/oxm"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring-beans.xsd

    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm.xsd">
```

目前可用的标签有以下这些：

- jaxb2-marshaller
- jibx-marshaller
- castor-marshaller

每一个标签将会在接下来的章节中逐一介绍。下面是一个 JAXB2 的配置示例：

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springfra
mework.ws.samples.airline.schema"/>
```

17.5 JAXB

JAXB 绑定编译器将 W3C XML 架构实现为一到数个 Java 类，一个 jaxb.properties 文件，可能还会有数个资源文件。JAXB 同时还支持从被注解的 Java 类生成 XML 架构。

Spring 支持基于 17.2 Marshaller 和 Unmarshaller 所提到的 Marshaller 和 Unmarshaller 结构实现的 JAXB 2.0 API 编组策略。相应的类文件都定义在 org.springframework.oxm.jaxb 包下面。

17.5.1 Jaxb2Marshaller

Jaxb2Marshaller 类同时实现了 Marshaller 和 Unmarshaller 接口。这个类需要上下文路径以正常运作，你可以通过 contextPath 属性来设置。上下文路径是一组由冒号（：）分隔的 Java 包名。这些包下面包含了由 XML 架构所生成的对应 Java 类。另外你可以通过设置一个叫 classesToBeBound 的属性来配置一组可以被编组器支持的类。架构的验证则通过向 bean 中配置一到多个 XML 架构的 xsd 文件资源来实现。下面是一个 bean 的配置示例：

```
<beans>
    <bean id="jaxb2Marshaller" class="org.springframework.oxm.ja
xb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>org.springframework.oxm.jaxb.Flight</valu
e>
                <value>org.springframework.oxm.jaxb.Flights</val
ue>
            </list>
        </property>
        <property name="schema" value="classpath:org/springframe
work/oxm/schema.xsd"/>
    </bean>
...
</beans>
```

17.5.2 基于 XML 架构的配置

Jaxb2-marshaller 标签 配置了一个 org.springframework.oxm.jaxb.Jaxb2Marshaller 实例。以下是一个配置实例：

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>
```

如果要配置需要被绑定的类，则可以使用 class-to-be-bound 子标签：

```
<oxm:jaxb2-marshaller id="marshaller">
    <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Airport"/>
    <oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Flight"/>
    ...
</oxm:jaxb2-marshaller>
```

可用的标签属性如下表：

属性	描述	是否必需
id	编组器的id	no
contextPath	JAXB上下文路径	no

17.6 Castor

Castor XML 映射是一个开源的 XML 绑定框架。它允许使用者将包含在一个 Java 对象模型中的数据转换成 XML 文档，或者反之。Castor 默认并不需要额外的配置就可以使用。但如果使用者希望能够对框架行为拥有更多的控制，则可以通过在配置中引入一个映射文件来达到这一目的。

读者可以从 Castor 的项目主页上了解更多相关内容。Spring 对这一框架的集成代码都在 org.springframework.oxm.castor 包底下。

17.6.1 CastorMarshaller

与 JAXB 类似，CastorMarshaller 类同时实现了 Marshaller 和 Unmarshaller 接口。它可以通过以下配置来被引用：

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.c
astor.CastorMarshaller" />
    ...
</beans>
```

17.6.2 映射

尽管 Castor 的默认编组行为基本能够适应大部分应用场景，有时候还是需要一些自定义的能力。这一需求可以通过使用一个 Castor 的映射文件来实现。更多信息请参考 Castor XML Mapping。

映射文件可以通过 mapppingLocation 属性进行设置，如下是一个配置了 classpath 资源的示例：

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.c
astor.CastorMarshaller" >
        <property name="mappingLocation" value="classpath:mappin
g.xml" />
    </bean>
</beans>
```

17.6.3 基于 XML 架构的配置

一个 castor-marshaller 代表了一个 org.springframework.oxm.castor.CastorMarshaller 实例。示例如下：

编组器实例可以用两种方式进行配置，一种是（通过 mapping-location 属性）指定映射文件的位置 ，另一种则是（通过 target-class 或 target-package 属性）指定包含有相应 XML 描述信息的 Java POJO。后一种方式一般会与通过 XML 架构自动生成 Java 代码的功能结合使用。

以下是可用的标签属性：

属性	描述	是否必需
id	编组器的id	no
encoding	反编组 XML 文件使用的编码	no
target-class	一个 Java POJO 类名，此类中包含一个（由代码自动生成器生成的） XML 类的描述信息	no
target-package	一个包含了（由代码自动生成器生成的）POJO 和相应 Castor XML 描述类的 Java 包名	no
mapping-location	Castor XML 映射文件的位置	no

17.7 JiBX

JiBX 框架提供的解决方案思路与 Hibernate 对于 ORM 的解决方案思路类似：通过一个绑定定义指定了你的 Java 对象与 XML 文件之间互相转换的规则。在准备好绑定并编译了类文件后，一个 JiBX 编译器将会对编译好的类文件进行增强，在其中加入一些辅助代码，并自动添加用于处理在类实例与 XML 文档之间相互转换的操作代码。

请参考 JiBX官方网站 来了解更多信息。Spring 对于框架的集成代码则都在 org.springframework.oxm.jibx 包下面。

17.7.1 JibxMarshaller

JiBXMarshaller 类同时实现了 Marshaller 和 Unmarshaller 接口。它需要使用者设置编组的目的类的类名才能正确工作。设置类名的属性是 targetClass。另外还有一个可选属性是 bingdingName，用户可以通过这个属性配置绑定名。接下来的示例中，我们将绑定 Flight 类：

```
<beans>
    <bean id="jibxFlightsMarshaller" class="org.springframework.
oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jib
x.Flights</property>
    </bean>
    ...
</beans>
```

一个 JibxMarshaller 只能处理一个目的类与 XML 的相互转换，如果你需要编组多个类，你必需配置相应数量的 JibxMarshallers bean，然后在 targetClass 里面指定相应各个类的类名。

17.7.2 基于 XML 的配置

jibx-marshaller 标签配置了 org.springframework.oxm.jibx.JibxMarshaller 的实例。以下是一个示例：

```
<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airline.schema.Flight"/>
```

标签的可用属性如下：

属性	描述	是否必需
id	编组器的id	no
target-class	此编组器所对应的目标类	yes
bindingName	此编组器使用的绑定名	no

17.8 XStream

Xstream 是一个用于将对象与 XML 文档进行序列化与反序列化的简单类库。它不需要任何映射关系，并且会生成整齐的 XML 文档。

请参考 XStream 的项目主页以获取更多信息。Spring 对此框架的集成代码都在 org.springframework.oxm.xstream 包下面。

XstreamMarshaller 类不需进行任何配置便可直接在 applicationContext.xml 文件中直接配置成 bean 进行使用。不过你可以配置一个包含了字符串别名与类之间对应关系的 别名映射表 来实现对 XML 解析结果的自定义：

```
<beans>
    <bean id="xstreamMarshaller" class="org.springframework.oxm.
xstream.XStreamMarshaller">
        <property name="aliases">
            <props>
                <prop key="Flight">org.springframework.oxm.xstre
am.Flight</prop>
            </props>
        </property>
    </bean>
    ...
</beans>
```

Xstream 默认允许对任何类进行反编组操作，但这可能会导致安全隐患。因此不建议使用 XStreamMarshaller 对来源于外部（比如公网）的 XML 文档进行反编组。如果你坚持使用 XStreamMarshaller 反编组来自外部的 XML 文档，请如下例演示的一样设置 supportedClasses 属性：

```
<bean id="xstreamMarshaller" class="org.springframework.oxm.xstr
eam.XStreamMarshaller">
    <property name="supportedClasses" value="org.springframework
.oxm.xstream.Flight"/>
    ...
</bean>
```

设置这一属性能够确保只有指定的类才能够被用于反编组。

更进一步，你可以通过注册 自定义转换器) 来确保只有你指定的类才能够被反编组。建议在明确指定了所有转换器后，在列表的最后加上 CatchAllConverter ，这样一来便可确保具有低优先级以及安全风险的 Xstream 默认转换器不会被调用。

这里需要注意的是 XStream 是一个 XML 序列化类库，而非一个数据绑定类库，所以它对命名空间的支持有限。这就使得 XStream 并不适合于用在网络服务中。

19.1 简介

Spring架构与其他MVC框架所不同的重要一点是视图技术，比如，决定使用Groovy Markup Templates 或者Thymeleaf代替JSP仅仅是配置的问题．这个章节主要设计主流的视图技术，以及简单提及怎样使用新的技术。这个章节假设你已经熟悉第18.5节“Resolving views” ，该章节涵盖了视图怎样与MVC框架结合的基本知识。

19.2 Thymeleaf

Thymeleaf是一种与MVC架构结合很好的视图技术. 不仅仅Spring团队而且Thymeleaf 自身也提供了很好的支持。

配置Thymeleaf 对Spring的支持通常需要定义几个bean, 像 `ServletContextTemplateResolver` , `SpringTemplateEngine` 和 `ThymeleafViewResolver` 。如需要更多详细信息请点击文档 Thymeleaf+Spring。

19.3 Groovy Markup Templates

Groovy Markup Template Engine 是另一种被Spring支持的视图技术,此模板引擎是一种主要用于生成类似XML的标记（XML，XHTML，HTML5，...）的模板引擎，但可用于生成任何基于文本的内容。

这需要在classpath上配置Groovy 2.3.1+。

19.3.1 配置

配置 Groovy Markup Template Engine相当容易:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

       @Override
    public void configureViewResolvers(ViewResolverRegistry regi
stry) {
        registry.groovy();
    }

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConf
igurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

使用MVC命名空间的XML文本:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:groovy/>
</mvc:view-resolvers>

<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>
```

19.3.2 例子

和传统模板引擎不同，这一个依赖于使用构建器语法的DSL。 以下是HTML页面的示例模板:

```
yieldUnescaped '<!DOCTYPE html>'
html(lang:'en') {
    head {
        meta('http-equiv':'"Content-Type" content="text/html; ch
arset=utf-8"')
        title('My page')
    }
    body {
        p('This is an example of HTML contents')
    }
}
```

19.4 FreeMarker

FreeMarker是一种模板语言，可以用作Spring MVC应用程序中的视图技术. 更多关于模板语言的信息，请点击站点 FreeMarker .

19.4.1 依赖

您的Web应用程序需要包含freemarker-2.x.jar才能使用FreeMarker。 通常这包含在WEB-INF / lib文件夹中，其中jar保证由Java EE服务器找到并添加到应用程序的类路径中。 当然，假设你已经有spring-webmvc.jar在你的'WEB-INF / lib'目录了！

19.4.2 上下文配置

通过将相关的configurer bean定义添加到您的'* -servlet.xml'来初始化一个合适的配置，如下所示：

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servl
et.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemark
er/"/>
</bean>

<!--
View resolvers can also be configured with ResourceBundles or XM
L files. If you need
different view resolving based on Locale, you have to use the re
source bundle resolver.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.v
iew.freemarker.FreeMarkerViewResolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".ftl"/>
</bean>
```

> 对于非web项目，在你定义的上下文中增加 `FreeMarkerConfigurationFactoryBean`

19.4.3 创建模板

您的模板需要存储在上面显示的FreeMarkerConfigurer指定的目录中。 如果您使用突出显示的视图解析器，则逻辑视图名称与模板文件名称以类似于JSP的InternalResourceViewResolver的方式相关。 因此，如果您的控制器返回一个包含视图名称为“welcome”的ModelAndView对象，则解析器将查找/WEB-INF/freemarker/welcome.ftl模板。

19.4.4 高级FreeMarker配置

FreeMarker的’Settings’和’SharedVariables’可以直接传递给由Spring管理的FreeMarker Configuration对象，方法是在FreeMarkerConfigurer bean上设置相应的bean属性。 freemarkerSettings属性需要一个java.util.Properties对象，而freemarkerVariables属性需要一个java.util.Map。

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
    <property name="freemarkerVariables">
        <map>
            <entry key="xml_escape" value-ref="fmXmlEscape"/>
        </map>
    </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

> 有关设置和变量的详细信息，请参阅FreeMarker文档，因为它们适用于Configuration对象。

19.4.5绑定支持和表单处理

Spring提供了一个用于JSP的标签库，其中包含（其中包含）<spring：bind />标签。 此标记主要允许表单从表单支持对象显示值，并显示来自网络或业务层的验证程序的失败验证结果。 Spring还支持FreeMarker中的相同功能，还有其他方便的宏用于生成表单输入元素。

绑定宏

在双方语言的spring-webmvc.jar文件中都保留了一组标准的宏，因此它们始终可供适当配置的应用程序使用。

在Spring库中定义的一些宏被认为是内部的（私有的），但在宏定义中不存在这样的范围，使所有的宏都可以被调用代码和用户模板。 以下部分仅出现在您在模板中直接调用的宏。 如果您希望直接查看宏代码，则该文件在包org.springframework.web.servlet.view.freemarker中称为spring.ftl。

简单绑定

在作为Spring MVC控制器的窗体视图的HTML表单（vm / ftl模板）中，您可以使用与以下类似的代码绑定到字段值，并以与JSP等效的类似方式显示每个输入字段的错误消息。 下面列出了以前配置的personForm视图的示例代码：

```
<!-- freemarker macros have to be imported into a namespace. We
strongly
recommend sticking to 'spring' -->
<#import "/spring.ftl" as spring/>
<html>
    ...
    <form action="" method="POST">
        Name:
        <@spring.bind "myModelObject.name"/>
        <input type="text"
            name="${spring.status.expression}"
            value="${spring.status.value?html}"/><br>
        <#list spring.status.errorMessages as error> <b>${error}
</b> <br> </#list>
        <br>
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

> <@ spring.bind>需要一个“path”参数，它由命令对象的名称组成（除非您在FormController属性中更改它，否则将是“command”），后跟一个句点和该命令上的字段名称 您要绑定到的对象。 也可以使用嵌套字段，如“command.address.street”。 bind macro 假定web.xml中ServletContext参数defaultHtmlEscape指定的默认HTML转义行为。
>
> 称为<@ spring.bindEscaped>的宏的可选格式需要第二个参数，并显式指定是否在状态错误消息或值中使用HTML转义。 根据需要设置为true或false。 附加的表单处理宏简化了HTML转义的使用，并且尽可能使用这些宏。 他们将在下一节中进行说明。

表单输入生成宏

两种语言的其他便利宏简化了绑定和表单生成（包括验证错误显示）。 从来没有必要使用这些宏来生成表单输入字段，并且可以混合使用简单的HTML或直接调用前面突出显示的弹簧绑定宏。

可用宏的下表显示了VTL和FTL定义以及每个参数列表。

宏	**FTL** 定义	消息（根据代码 束输出字符串）
<@spring.message code/>	messageText（根据代码参数从资源束输出一个字符串，退回到默认参数的值）	<@spring.mess code, text/>
url(使用应用程序的上下文根前缀相对URL)	<@spring.url relativeUrl/>	formInput（用 输入的标准输入
<@spring.formInput path, attributes, fieldType/>	**formHiddenInput*** (用于提交非用户输入的隐藏输入字段)	<@spring.formI path, attributes/
formPasswordInput* (用于收集密码的标准输入字段。请注意，不会在此类型的字段中填充任何值)	<@spring.formPasswordInput path, attributes/>	**formTextarea**(用于收集长，自 本输入)
<@spring.formTextarea path, attributes/>	**formSingleSelect**(d下拉框选项允许选择单个所需的值)	<@spring.formS path, options, a
formMultiSelect(允许用户选择0个或更多值的选项列表框)	<@spring.formMultiSelect path, options, attributes/>	**formRadioButt** 选按钮允许从可 行单次选择)
<@spring.formRadioButtons path, options separator, attributes/>	**formCheckboxes**(一组允许选择0个或更多值的复选框)	<@spring.formC path, options, s attributes/>
formCheckbox(a single checkbox)	<@spring.formCheckbox path, attributes/>	**showErrors**(简 字段的验证错误

- 在FTL（FreeMarker）中，这两个宏并不是实际需要的，因为您可以使用正常的formInput宏，指定'hidden'或``password'作为`fieldType参数的值。.

任何上述宏的参数具有一致的含义:

- **path**： 要绑定的字段的名称（如："command.name"）
- 选项： 可以在输入字段中选择的所有可用值的映射。地图的键代表将从窗体返回并绑定到命令对象的值。按键显示的映射对象是表单上显示给用户的标签，可能与表单发回的对应值不同。通常这样的地图由控制器作为参考数据提供。可以根据需要的行为使用任何Map实现。对于严格排序的映射，可以使用诸如具有适当比较器的TreeMap的SortedMap，并且可以使用应该以插入顺序返回值的任意地图，使用commons-collections中的LinkedHashMap或LinkedMap
- 分隔符：多个选项可用作谨慎元素（单选按钮或复选框），用于分隔列表中每

个选项的字符序列（即“
”）. 属性：要包含在HTML标签本身中的任意标签或文本的附加字符串。该字符串由字符串回显。例如，在textarea字段中，您可以提供“rows =”5“cols =”60“’的属性，或者您可以传递样式信息，如”style =“border：1px solid silver”

- **classOrStyle**：对于showErrors宏，包含每个错误的span标签将使用的CSS类的名称。如果没有提供信息（或值为空），那么错误将被包裹在<b> </ b>标签中。

宏中的一些例子在FTL和VTL中的一些中概述。在两种语言之间存在使用差异的情况下，它们将在说明中进行说明。

输入的值

formInput宏在上面的示例中使用path参数（command.name）和一个空的附加属性参数。 宏与所有其他表单生成宏一起在路径参数上执行隐式弹簧绑定。 绑定保持有效，直到发生新的绑定，因此showErrors宏不需要再次传递路径参数 – 它只是在上次创建绑定的字段时操作。

showErrors宏采用一个分隔符参数（用于在给定字段上分隔多个错误的字符），并且还接受第二个参数，此时为类名或样式属性。 请注意，FreeMarker能够为attributes参数指定默认值。

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

输出显示在生成名称字段的表单片段中，并在表单提交后在该字段中没有值显示验证错误。 验证通过Spring的验证框架进行。

生成的HTML如下所示：

```
Name:
<input type="text" name="name" value="">
<br>
    <b>required</b>
<br>
<br>
```

formTextarea宏的工作方式与formInput宏相同，并接受相同的参数列表。 通常，第二个参数（属性）将用于传递文本区域的样式信息或行和列属性。

选择字段

四个选择字段宏可用于在HTML表单中生成常见的UI值选择输入。

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes四个宏中的每一个接受一个包含表单字段的值的选项映射，以及与该值对应的标签。 该值和标签可以相同。FTL中的单选按钮示例如下。表单后备对象为此字段指定了“伦敦”的默认值，因此不需要进行验证。 当表单呈现时，可以在模型中以“cityMap”的名称提供作为参考数据的整个城市列表。

```
...
Town:
<@spring.formRadioButtons "command.address.town", cityMap, ""/><
br><br>
```

这将使用一个单选按钮，一个为cityMap中的每个值使用分隔符“”。 不提供其他属性（缺少宏的最后一个参数）。 cityMap对地图中的每个键值对使用相同的String。地图的键是表单实际提交的POST请求参数，地图值是用户看到的标签。 在上面的例子中，给出了三个众所周知城市的列表和表单后备对象中的默认值，HTML将是

```
Town:
<input type="radio" name="address.town" value="London">London</i
nput>
<input type="radio" name="address.town" value="Paris" checked="c
hecked">Paris</input>
<input type="radio" name="address.town" value="New York">New Yor
k</input>
```

如果您的应用程序期望通过内部代码处理城市，则将使用如下面的示例的合适键创建代码映射。

```
protected Map<String, String> referenceData(HttpServletRequest r
equest) throws Exception {
    Map<String, String> cityMap = new LinkedHashMap<>();
    cityMap.put("LDN", "London");
    cityMap.put("PRS", "Paris");
    cityMap.put("NYC", "New York");

    Map<String, String> model = new HashMap<>();
    model.put("cityMap", cityMap);
    return model;
}
```

代码根据单选框的相关代码产生新的输出，但用户仍然会看到更友好的城市名称。

```
Town:
<input type="radio" name="address.town" value="LDN">London</inpu
t>
<input type="radio" name="address.town" value="PRS" checked="che
cked">Paris</input>
<input type="radio" name="address.town" value="NYC">New York</in
put>
```

HTML转义和符合XHTML标准

上述格式宏的默认使用将导致符合HTML 4.01的HTML标记，并且使用Spring绑定支持使用的web.xml中定义的HTML转义的默认值。 为了使标签符合XHTML或覆盖默认的HTML转义值，您可以在模板中指定两个变量（或者在模型中指定模板中可以看到的变量）。 在模板中指定它们的优点是，它们可以在模板处理中稍后更改为不同的值，以便为表单中的不同字段提供不同的行为。

要为您的标记切换到XHTML符合性，请为名为xhtmlCompliant的模型/上下文变量指定值“true”

```
<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

处理此指令后，Spring宏生成的任何标签现在都符合XHTML标准。

以类似的方式，可以为每个字段指定HTML转义：

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name"/>

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

19.5 JSP & JSTL

Spring为JSP和JSTL视图提供了几个开箱即用的解决方案。 使用JSP或JSTL是使用在WebApplicationContext中定义的常规视图解析器完成的。 此外，当然，您需要编写一些实际渲染视图的JSP。

> 设置应用程序使用JSTL是一个常见的错误源，主要是由于混淆了不同的servlet规范JSP和JSTL版本号，它们的含义和如何正确声明taglibs引起的。 文章
>
> How to Reference and Use JSTL in your Web Application
>
> 提供了常见的陷阱和如何避免它们的有用指南。 请注意，从Spring 3.0开始，最小支持的servlet版本为2.4（JSP 2.0和JSTL 1.1），这有助于减少混淆的范围。

19.5.1 视图解析

就像您与Spring集成的任何其他视图技术一样，对于JSP，您将需要一个视图解析器来解析你的视图。 使用JSP进行开发时最常用的视图解析器是InternalResourceViewResolver和ResourceBundleViewResolver。 两者都在WebApplicationContext中声明：

```
<!-- the ResourceBundleViewResolver -->
<bean id="viewResolver" class="org.springframework.web.servlet.v
iew.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-
INF/classes):
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlVie
w
productList.url=/WEB-INF/jsp/productlist.jsp
```

如您所见，ResourceBundleViewResolver需要一个属性文件来定义映射到1）一个类和2个URL的视图名称。 使用ResourceBundleViewResolver，您只能使用一个解析器来混合不同类型的视图。

```
<bean id="viewResolver" class="org.springframework.web.servlet.v
iew.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.se
rvlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

InternalResourceBundleViewResolver可以配置为如上所述使用JSP。 作为最佳做法，我们强烈建议您将JSP文件放在“WEB-INF”目录下的目录中，以免客户端直接访问。

19.5.2 ‘Plain-old’ JSPs 与JSTL

当使用Java标准标签库时，必须使用特殊的视图类JstlView，因为JSTL需要一些准备工作，例如I18N功能才能正常工作。

19.5.3 Additional tags facilitating development

Spring提供了请求参数到命令对象的数据绑定，如前几章所述。 为了方便JSP页面的开发与这些数据绑定功能的结合，Spring提供了一些使事情更容易的标签。 所有Spring标签都具有HTML转义功能，以启用或禁用字符转义。

标签库描述符（TLD）包含在spring-webmvc.jar中。 有关各个标签的更多信息，请参见附录“???”。

19.5.4 使用Spring的表单标签库

从版本2.0开始，Spring提供了一套全面的数据绑定感知标签，用于在使用JSP和Spring Web MVC时处理表单元素。 每个标签提供对其对应的HTML标签对应的一组属性的支持，使标签熟悉和直观地使用。 标记生成的HTML符合HTML 4.01 / XHTML 1.0标准。

与其他表单/输入标签库不同，Spring的表单标签库与Spring Web MVC集成，使标签能够访问控制器处理的命令对象和引用数据。 如以下示例所示，表单标签使JSP更易于开发，阅读和维护。

我们来看看表单标签，看一下每个标签的使用方式。 我们已经包括生成的HTML片段，其中某些标签需要进一步的讨论。

配置

表单标签库在spring-webmvc.jar中附带。 库描述符称为spring-form.tld。要使用此库中的标记，请将以下指令添加到JSP页面的顶部：

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

其中form是要用于此库的标签的标签名称前缀。

表单标签

此标记呈现HTML“form”标签，并将绑定路径公开到内部标签以进行绑定。 它将命令对象放在PageContext中，以便可以通过内部标记访问命令对象。 该库中的所有其他标签都是表单标签的嵌套标签。

假设我们有一个名为User的域对象。 它是一个具有诸如firstName和lastName等属性的JavaBean。 我们将使用它作为返回form.jsp的表单控制器的表单支持对象。 下面是一个form.jsp的例子：

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

firstName和lastName值由页面控制器从PageContext中放置的命令对象中检索。继续阅读以查看更多关于内部标签如何与表单标签一起使用的复杂示例。

生成的HTML看起来像一个标准格式：

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="Harry
"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="Potter
"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

前面的JSP假定表单后备对象的变量名是'command'。如果您将表单支持对象放在另一个名称下的模型中（绝对是最佳实践），那么可以将表单绑定到命名变量，如下所示：

```
<form:form modelAttribute="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

输入标签

此标记默认使用绑定值和type ='text'呈现HTML“输入”标签。 有关此标记的示例，请参阅“The form tag”一节。 从Spring 3.1开始，您可以使用其他类型的HTML5特定类型，如“email”，“tel”，“date”等。

勾选标签

此标记会显示一个类型为“checkbox”的HTML“input”标签。让我们假设我们的用户有喜好，如通讯订阅和爱好列表。 以下是Preferences类的示例：

```
public class Preferences {

    private boolean receiveNewsletter;
    private String[] interests;
    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter)
{
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

The form.jsp would look like:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <%-- Approach 2: Property is of an array or of type java.util.Collection --%>
            <td>
                Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path="preferences.interests" value="Defence Against the Dark Arts"/>
            </td>
        </tr>

        <tr>
            <td>Favourite Word:</td>
            <%-- Approach 3: Property is of type java.lang.Object --%>
            <td>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>
```

复选框标签有3种方法可以满足您所有的复选框需求。

- 方法一 – 当绑定值的类型为java.lang.Boolean时，如果绑定值为true，则输入（复选框）将被标记为“已检查”。 valueattribute对应于setValue（Object）value属性的已解析值。
- 方法二 – 当绑定值为array或java.util.Collection类型时，如果配置的setValue（Object）值存在于绑定集合中，则输入（复选框）将被标记为“checked”.
- 方法三 – 对于任何其他绑定值类型，如果配置的setValue（Object）等于绑定值，则输入（复选框）将被标记为“已检查”。

请注意，无论采用何种方法，都会生成相同的HTML结构。 以下是某些复选框的HTML片段：

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="che
ckbox" value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.intere
sts"/>
        Herbology: <input name="preferences.interests" type="che
ckbox" value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.intere
sts"/>
        Defence Against the Dark Arts: <input name="preferences.
interests" type="checkbox" value="Defence Against the Dark Arts"
/>
        <input type="hidden" value="1" name="_preferences.intere
sts"/>
    </td>
</tr>
```

您可能不会看到的是每个复选框后面的额外的隐藏字段。 当HTML页面中的复选框未被选中时，一旦表单被提交，它的值就不会作为HTTP请求参数的一部分发送到服务器，所以我们需要一个解决方法来为HTML解决这个问题，以便Spring表单数据绑定 工作。 复选框标记遵循现有的Spring约定，为每个复选框添加一个前缀为下划线（“_”）的隐藏参数。 通过这样做，您正在有效地告诉Spring，“复选框在表单中可见，并且我希望我的对象将窗体数据绑定到其中，以反映该复选框的状态，无论什么”。

复选框标记

此标记会使用“checkbox”类型呈现多个HTML“input”标签。

基于上一个复选框标记部分的示例。 有时您不希望在JSP页面中列出所有可能的选项。 您希望在运行时提供可用选项的列表，并将其传递给标签。 这就是复选框标签的目的。 您传递一个数组，列表或地图，其中包含“items”属性中的可用选项。通常，bound属性是一个集合，因此它可以保存用户选择的多个值。 以下是使用此标记的JSP的示例：

```
<form:form>
    <table>
        <tr>
            <td>Interests:</td>
            <td>
                <%-- Property is of an array or of type java.uti
l.Collection --%>
                <form:checkboxes path="preferences.interests" it
ems="${interestList}"/>
            </td>
        </tr>
    </table>
</form:form>
```

此示例假定“兴趣列表”是可用作包含要从中选择的值的字符串的模型属性的列表。在使用地图的情况下，将使用地图条目键作为值，并将地图条目的值用作要显示的标签。 您还可以使用自定义对象，您可以在其中使用“itemValue”提供值的属性名称，并使用“itemLabel”提供标签。

单选标记

此标记呈现类型为“radio”的HTML“input”标签。典型的使用模式将涉及绑定到相同属性但具有不同值的多个标签实例。

```
<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/>
    </td>
</tr>
```

多个单选框标签

此标签使用“radio”类型呈现多个HTML“input”标签。

就像上面的复选框标签一样，您可能希望将可用选项作为运行时变量传递。 对于这种用法，您将使用radiobuttons标签。 您传递一个数组，列表或地图，其中包含“items”属性中的可用选项。 在使用地图的情况下，将使用地图条目键作为值，并将地图条目的值用作要显示的标签。 您还可以使用自定义对象，您可以在其中使用“itemValue”提供值的属性名称，并使用“itemLabel”提供标签。

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></t
d>
</tr>
```

密码标签

此标记使用绑定值呈现类型为“password”的HTML“input”标签。

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password"/>
    </td>
</tr>
```

请注意，默认情况下，密码值未显示。 如果您想要显示密码值，则将“showPassword”属性的值设置为true，如此。

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showP
assword="true"/>
    </td>
</tr>
```

选择标签

此标记呈现HTML“选择”元素。 它支持与选定选项的数据绑定以及嵌套选项和选项标签的使用。

假设用户有一个技能列表。

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
</tr>
```

如果用户的技能在Herbology，“技能”行的HTML源代码将如下所示：

```
<tr>
    <td>Skills:</td>
    <td>
        <select name="skills" multiple="true">
            <option value="Potions">Potions</option>
            <option value="Herbology" selected="selected">Herbol
ogy</option>
            <option value="Quidditch">Quidditch</option>
        </select>
    </td>
</tr>
```

可选 标签

此标记呈现HTML“选项”。 它根据绑定值适当地设置“选择”。

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

如果用户的房子在Gryffindor，“房子”行的HTML源代码将如下所示：

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="selected">Gryff
indor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
</tr>
```

多个可选标签

此标记呈现HTML“option”标签的列表。 它根据绑定值适当地设置'selected'属性。

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code
" itemLabel="name"/>
        </form:select>
    </td>
</tr>
```

如果用户居住在英国，“Country”行的HTML源代码将如下所示：

```
<tr>
    <td>Country:</td>
    <td>
        <select name="country">
            <option value="-">--Please Select</option>
            <option value="AT">Austria</option>
            <option value="UK" selected="selected">United Kingdo
m</option>
            <option value="US">United States</option>
        </select>
    </td>
</tr>
```

如示例所示，选项标签与选项标签的组合使用将生成相同的标准HTML，但允许您显式指定JSP中仅用于显示（在其所在的位置）的值，例如 示例：“ - 请选择”。

items属性通常用一组或多个项目对象填充。 itemValue和itemLabel只是指定这些项目对象的bean属性; 否则，项目对象本身将被字符串化。 或者，您可以指定项目地图，在这种情况下，地图键将被解释为选项值，地图值对应于选项标签。 如果同时指定了itemValue和/或itemLabel，则item value属性将应用于map键，item label属性将应用于map值。

textarea标签

此标记呈现HTML“textarea”。

```
 <tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20"/></td>
    <td><form:errors path="notes"/></td>
</tr>
```

hidden 标签

此标记使用绑定值呈现类型为“hidden”的HTML“input”标签。 要提交未绑定的隐藏值，请使用类型为“hidden”的HTML输入标签。

<form：hidden path =“house”/>
如果我们选择将“房屋”值提交为隐藏值，则HTML将如下所示：

```
<input name="house" type="hidden" value="Gryffindor"/>
```

errors 标签

此标记会在HTML’span’标签中呈现字段错误。 它提供对控制器中创建的错误的访问或由与控制器相关联的任何验证器创建的错误。

假设我们要在提交表单时显示firstName和lastName字段的所有错误消息。 我们有一个名为UserValidator的User类的实例的验证器。

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "first
Name", "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastN
ame", "required", "Field is required.");
    }
}
```

form.jsp如下所示：

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName"/></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

如果我们在firstName和lastName字段中提交一个带有空值的表单，这就是HTML的样子：

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></
td>
            <%-- Associated errors to firstName field displayed
--%>
            <td><span name="firstName.errors">Field is required.
</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></t
d>
            <%-- Associated errors to lastName field displayed -
-%>
            <td><span name="lastName.errors">Field is required.<
/span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

如果我们要显示给定页面的整个错误列表怎么办？下面的示例显示了errors标签还支持一些基本的通配符功能。

- `path="*"` – 显示所有的错误
- `path="lastName"` – 显示与 `lastName` 相关的错误
- `假如path` 省略- 仅显示对象错误

下面的示例将显示页面顶部的错误列表，后跟字段旁边的字段特定错误:

```
<form:form>
    <form:errors path="*" cssClass="errorBox"/>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <td><form:errors path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

HTML是这样的：

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br
/>Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></
td>
            <td><span name="firstName.errors">Field is required.
</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></t
d>
            <td><span name="lastName.errors">Field is required.<
/span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
       </table>
</form>
```

HTTP 方法转变

REST的一个关键原则是使用统一接口。这意味着可以使用相同的四种HTTP方法来操作所有资源（URL）：GET，PUT，POST和DELETE。对于每个方法，HTTP规范定义了准确的语义。例如，GET应该永远是一个安全的操作，这意味着没有副作用，PUT或DELETE应该是幂等的，这意味着你可以一遍又一遍地重复这些操作，但最终结果应该是一样的。虽然HTTP定义了这四种方法，但HTML只支持两种方法：GET和POST。幸运的是，有两种可能的解决方法：您可以使用JavaScript来执行您的PUT或DELETE，或者使用“real”方法作为附加参数（在HTML表单中建模为隐藏输入字段）进行POST。后一个技巧是Spring的HiddenHttpMethodFilter所做

的。此过滤器是一个简单的Servlet过滤器，因此可以与任何Web框架（不仅仅是Spring MVC）结合使用。只需将此过滤器添加到您的web.xml中，并将带有隐藏_method参数的POST转换为相应的HTTP方法请求。

为了支持HTTP方法转换，Spring MVC表单标记已更新，以支持设置HTTP方法。例如，从更新的Petclinic示例中获取以下代码段

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/><
/p>
</form:form>
```

这将实际执行HTTP POST，隐藏在请求参数之后的“real”DELETE方法，由HiddenHttpMethodFilter拾取，如web.xml中所定义：

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMetho
dFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

相应的@Controller方法如下所示：

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable
 int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

HTML5 标签

从Spring 3开始，Spring表单标签库允许输入动态属性，这意味着您可以输入任何特定于HTML5的属性。

在Spring 3.1中，表单输入标签支持输入“text”以外的类型属性。 这旨在允许渲染新的HTML5特定输入类型，如 ’email’, ‘date’, ‘range’等。 请注意，输入type =’text’不是必需的，因为’text’是默认类型。

19.6 Script 模板

可以使用Spring将任何在JSR-223脚本引擎上运行的模板库集成到Web应用程序中。以下以广泛的方式描述如何做到这一点。脚本引擎必须实现ScriptEngine和Invocable接口。

已通过以下测试：

- H andlebars running on Nashorn
- Mustache running on Nashorn
- React running on Nashorn
- EJS running on Nashorn
- ERB running on JRuby
- String templates running on Jython

19.6.1依赖关系

为了能够使用脚本模板集成，您需要在您的类路径中提供脚本引擎：

- Nashorn Javascript引擎内置Java 8+。强烈建议使用最新的更新版本。
- Rhino Javascript引擎内建Java 6和Java 7.请注意，不建议使用Rhino，因为它不支持运行大多数模板引擎。
- 应该添加JRuby依赖以获得Ruby的支持。
- 应该添加Jython依赖关系，以获得Python的支持。

您还需要为基于脚本的模板引擎添加依赖关系。例如，对于Javascript，您可以使用WebJars添加Maven / Gradle依赖关系，以使您的javascript库在类路径中可用。

19.6.2如何集成基于脚本的模板

为了能够使用脚本模板，您必须对其进行配置，以便指定各种参数，如要使用的脚本引擎，要加载的脚本文件以及应该调用哪些函数来呈现模板。这是由于ScriptTemplateConfigurer bean和可选的脚本文件。

例如，为了渲染Mustache模板，感谢Java 8+提供的Nashorn Javascript引擎，您应该声明以下配置：

```
@Configuration
@EnableWebMvc
public class MustacheConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry regi
stry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplate
Configurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

XML 表示如下:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:script-template/>
</mvc:view-resolvers>

<mvc:script-template-configurer engine-name="nashorn" render-obj
ect="Mustache" render-function="render">
    <mvc:script location="mustache.js"/>
</mvc:script-template-configurer>
```

你期望的controller：

```
@Controller
public class SampleController {

    @RequestMapping
    public ModelAndView test() {
        ModelAndView mav  = new ModelAndView();
        mav.addObject("title", "Sample title").addObject("body",
 "Sample body");
        mav.setViewName("template.html");
        return mav;
    }
}
```

而Mustache模板是：

```
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <p>{{body}}</p>
    </body>
</html>
```

使用以下参数调用render函数：

- 字符串模板：模板内容
- 地图模型：视图模型
- String url：模板url（自4.2.2起）

Mustache.render（）与此签名本身兼容，因此您可以直接调用它。

如果您的模板技术需要一些自定义，您可以提供一个实现自定义渲染功能的脚本。例如，Handlerbars需要在使用它们之前编译模板，并且需要一个polyfill才能模拟服务器端脚本引擎中不可用的一些浏览器设施。

```
@Configuration
@EnableWebMvc
public class MustacheConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry regi
stry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplate
Configurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "r
ender.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}
```

> [注意]
>
> 如果使用非线程安全的脚本引擎，而不使用非并行设计的模板库（例如，在Nashorn上运行的Handlebars或React），则需要将sharedEngine属性设置为false。在这种情况下，由于这个错误，需要Java 8u60或更高版本。

polyfill.js仅定义Handlebars需要正确运行的窗口对象：

```
var window = {};
```

这个基本的render.js实现在使用之前编译模板。生产就绪实现还应该存储和重新使用缓存的模板/预编译模板。这可以在脚本端完成，以及您需要的任何定制（例如，管理模板引擎配置）。

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

19.7 XML编组视图

MarshallingView使用org.springframework.oxm包中定义的XML Marshaller将响应内容呈现为XML。 要编组的对象可以使用MarhsallingView的`modelKey bean属性显式设置。 或者，视图将遍历所有模型属性并编组Marshaller支持的第一个类型。有关org.springframework.oxm软件包中的功能的更多信息，请参阅使用O / X Mappers编组XML的章节。

19.8 Tiles

在使用Spring的Web应用程序中，可以将Tiles 与任何其他视图技术集成在一起。以下以广泛的方式描述如何做到这一点。

19.8.1 依赖

为了可以使用Tiles，您必须添加对Tiles 3.0.1版或更高版本的依赖关系，以及其对您的项目的传递依赖性。

19.8.2 怎样集成 Tiles

为了使用Tiles，你需要使用文件定义 配置 (对于基本定义和其他Tiles内容，请查看http://tiles.apache.org). 在Spring 是使用 `TilesConfigurer` . 请看下面的例子:

```
<bean id="tilesConfigurer" class="org.springframework.web.servle
t.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>
</bean>
```

您可以看到，有五个文件包含定义，它们都位于"WEB-INF / defs"目录中。 在WebApplicationContext初始化时，将加载文件，并初始化定义工厂。 之后，这个定义文件中的Tile可以在Spring Web应用程序中用作视图。 为了能够使用视图，您必须拥有一个ViewResolver，就像Spring使用的任何其他视图技术一样。 下面你可以找到两种可能性：UrlBasedViewResolver和ResourceBundleViewResolver。

您可以通过添加下划线，然后添加区域设置来指定特定于区域设置的瓷砖定义。 例如：

```
<bean id="tilesConfigurer" class="org.springframework.web.servle
t.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/tiles.xml</value>
            <value>/WEB-INF/defs/tiles_fr_FR.xml</value>
        </list>
    </property>
</bean>
```

使用此配置，tiles_fr_FR.xml将用于具有fr_FR语言环境的请求，默认情况下将使用tiles.xml。

UrlBasedViewResolver

UrlBasedViewResolver为给定的viewClass实例化其必须解析的每个视图。

```
<bean id="viewResolver" class="org.springframework.web.servlet.v
iew.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.web.se
rvlet.view.tiles3.TilesView"/>
</bean>
```

ResourceBundleViewResolver

ResourceBundleViewResolver必须提供一个属性文件，其中包含解析器可以使用的viewnames和viewclasses：

```
<bean id="viewResolver" class="org.springframework.web.servlet.v
iew.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>
```

```
...
welcomeView.(class)=org.springframework.web.servlet.view.tiles3.
TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles3.Til
esView
vetsView.url=vetsView (again, this is the name of a Tiles defini
tion)

findOwnersForm.(class)=org.springframework.web.servlet.view.Jstl
View
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

您可以看到，当使用ResourceBundleViewResolver时，您可以轻松地混合不同的视图技术。

请注意，TilesView类支持JSTL（JSP标准标记库）。

SimpleSpringPreparerFactory and SpringBeanPreparerFactory

作为高级功能，Spring还支持两种特殊的Tiles PreparerFactory实现。有关如何在您的Tiles定义文件中使用ViewPreparer引用的详细信息，请查看Tiles文档。

指定SimpleSpringPreparerFactory根据指定的preparer类自动导航ViewPreparer实例，应用Spring的容器回调以及应用配置的Spring BeanPostProcessors。如果Spring的上下文范围注释配置已激活，ViewPreparer类中的注释将被自动检测并应用。请注意，这需要在Tiles定义文件中的preparer类，就像默认的PreparerFactory一样。

指定SpringBeanPreparerFactory对指定的prepareer名称而不是类进行操作，从DispatcherServlet的应用程序上下文中获取相应的Spring bean。在这种情况下，完整的bean创建过程将控制Spring应用程序上下文，允许使用显式依赖注入配置，范围bean等。请注意，您需要为每个preparer名称定义一个Spring bean定义（如你的Tiles定义）。

```
<bean id="tilesConfigurer" class="org.springframework.web.servle
t.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>

    <!-- resolving preparer names as Spring bean definition name
s -->
    <property name="preparerFactoryClass"
            value="org.springframework.web.servlet.view.tiles3.S
pringBeanPreparerFactory"/>

</bean>
```

20. CORS 支持

20.1 简介

出于安全考虑，浏览器禁止AJAX调用驻留在当前来源之外的资源。 例如，当您在一个标签中检查您的银行帐户时，您可以在另一个标签中打开evil.com网站。 evil.com的脚本不能使用您的凭据向您的银行API发出AJAX请求（例如，从您的帐户中提款）！

Cross-origin resource sharing(CORS) 是 大多数浏览器实现的W3C规范，允许您以灵活的方式指定什么样的跨域请求被授权，而不是使用一些较不安全和不太强大的黑客工具（如IFRAME或JSONP）。

从Spring Framework 4.2开始，CORS支持开箱即用。. CORS 请求(including preflight ones with an `OPTIONS` method) 被自动调度到各种已注册的HandlerMappings. 由于CorsProcessor的实现（默认为DefaultCorsProcessor），为了根据您提供的CORS配置添加相关的CORS响应头（如Access-Control-Allow-Origin），它们处理CORS预检请求并拦截CORS简单实际的请求。

20.2 Controller 方法CORS 配置

你如果想使用CORS ，可以在 `@RequestMapping上增加 @CrossOrigin` 注解. 默认情况下@CrossOrigin允许@RequestMapping注释中指定的所有起始点和HTTP方法：

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}
")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

对于整个 controller的CORS 的支持:

```
@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}
")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

在上面的例子中， `retrieve()` 和 `remove()对实现了对CORS 的支持` ,因此你可以使用 `@CrossOrigin` 属性对自已的程序进行自定义的CORS 配置。

您甚至可以同时使用控制器级和方法级的CORS配置; 然后，Spring将组合两个注释的属性以创建合并的CORS配置。

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("http://domain2.com")
    @RequestMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @RequestMapping(method = RequestMethod.DELETE, path = "/{id}
")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

20.3 全局CORS 配置

除了细粒度，基于注释的配置，您也可能想要定义一些全局CORS配置。 这与使用过滤器类似，但可以在Spring MVC中声明，并结合细粒度的@CrossOrigin配置。默认情况下，所有起始点和GET，HEAD和POST方法都是允许的。

20.3.1 JavaConfig

配置全局CORS 如下所示:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**");
    }
}
```

你也可以轻易的改变任意属性，以及仅将此CORS配置应用于特定路径模式：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(false).maxAge(3600);
    }
}
```

20.3.2 XML 命名空间

以下最小的XML配置为/ **路径模式启用CORS，具有与上述JavaConfig示例相同的默认属性：

```
<mvc:cors>
    <mvc:mapping path="/**" />
</mvc:cors>
```

也可以使用自定义属性声明几个CORS映射：

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="http://domain1.com, http://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="fa
lse"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="http://domain1.com" />

</mvc:cors>
```

20.4 高级自定义

CorsConfiguration 允许你定义CORS 请求怎样被处理: 允许origins, headers, methods, 如.：他可以使用以下几种方式处理:

- `AbstractHandlerMapping#setCorsConfiguration()` 允许指定映射到路径模式（如/ api / **）的几个 CorsConfiguration实例的Map
- 子类通过重载 `AbstractHandlerMapping#getCorsConfiguration(Object, HttpServletRequest)` 方法提供自己的 `CorsConfiguration` .
- Handlers 通过实现 `CorsConfigurationSource` 接口 (像 `ResourceHttpRequestHandler` 现在做的一样) 对每个请求提供一个CorsConfiguration 实例.

20.5 基于 CORS 支持的过滤

为了支持基于过滤器的安全框架（如Spring Security）的CORS，或者与其他不支持本地CORS的库一起支持CORS，Spring Framework还提供了一个 `CorsFilter` . 而不是使用@CrossOrigin或WebMvcConfigurer#addCorsMappings（CorsRegistry），您需要注册一个自定义过滤器，如下所示:

```
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSou
rce;
import org.springframework.web.filter.CorsFilter;

public class MyCorsFilter extends CorsFilter {

    public MyCorsFilter() {
        super(configurationSource());
    }

    private static UrlBasedCorsConfigurationSource configuration
Source() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("http://domain1.com");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        UrlBasedCorsConfigurationSource source = new UrlBasedCor
sConfigurationSource();
        source.registerCorsConfiguration("/**", config);
        return source;
    }
}
```

您需要确保CorsFilter在其他过滤器之前过滤, 请参考this blog post ，了解如何配置Spring Boot。

21. 与其他Web框架集成

21.1 简介

Spring Web Flow

Spring Web Flow (SWF) 旨在成为管理Web应用程序页面流的最佳解决方案。

SWF与Servlet和Portlet环境中的Spring MVC和JSF等现有框架集成。 如果您有一个业务流程（或流程）将受益于会话模型而不是纯粹的请求模型，则SWF可能是解决方案。

SWF允许您将逻辑页面流作为在不同情况下可重用的自包含模块捕获，因此非常适合构建引导用户通过驱动业务流程的受控导航的Web应用程序模块。

有关SWF的更多信息，请参阅Spring Web Flow website.

本章详细介绍了Spring与第三方Web框架的集成，如 JSF.

Spring框架的核心价值主张之一是自由选择。在一般意义上，Spring并不强迫某人使用或购买任何特定的架构，技术或方法（尽管它肯定建议其他人使用）。这种选择与开发人员及其开发团队最相关的架构，技术或方法的自由可以说是在Web领域最为明显的一个领域，Spring领域同时提供了自己的Web框架（Spring MVC）提供与许多受欢迎的第三方网络框架的集成。这允许人们继续利用在特定的Web框架（如JSF）中获得的任何和所有技能，同时能够享受Spring在其他领域（如数据访问，声明式交易）管理，灵活配置和应用程序组装。

放弃了the woolly sales patter（c.f.前一段），本章的其余部分将集中在Web框架与Spring集成的细节。开发人员从其他语言开始Java经常评论的一件事就是Java中可用的Web框架的丰富程度。 Java中确实有大量的Web框架;事实上，在一个章节中有太多的东西可以覆盖任何细节的外表。本章从Java中选出了四个更受欢迎的Web框架，从所有支持的Web框架通用的Spring配置开始，然后详细说明每个支持的Web框架的特定集成选项。

> 请注意，本章不会尝试解释如何使用任何受支持的Web框架。例如，如果要将JSF用于Web应用程序的表示层，那么假设您已经熟悉了JSF本身。如果您需要有关任何支持的Web框架本身的更多详细信息，请参阅本章末尾 Section 21.6, “Further Resources” .

21.2 普通配置

在介绍每个支持的Web框架的集成细节之前，我们首先来看看Spring配置，这些配置不是特定于任何一个Web框架的。（本节同样适用于Spring自己的Web框架，Spring MVC。）

（Spring’s）轻量级应用程序模型支持的一个概念（为了想要更好的词）就是分层架构。请记住，在“经典”分层架构中，网络层只是多层次之一;它作为服务器端应用程序的入口点之一，它委托服务层定义的服务对象（外观），以满足业务特定（和表示技术不可知）用例。在Spring中，这些服务对象，任何其他特定于业务的对象，数据访问对象等存在于不同的“业务上下文”中，其不包含web或表示层对象（诸如Spring MVC控制器之类的呈现对象通常在独特的“演示文稿”）。本节详细介绍了如何在一个应用程序中配置包含所有“business bean”的Spring容器（WebApplicationContext）。

具体细节：所有需要做的是在一个人的Web应用程序的标准Java EE servlet web.xml文件中声明一个ContextLoaderListener，并添加一个contextConfigLocation <context-param/>（在同一个文件中）来定义哪个集合的Spring XML配置文件加载。

在<listener />配置下方查找：

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoade
rListener</listener-class>
</listener>
```

在 <context-param/> 配置下方查找：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

如果不指定contextConfigLocation上下文参数，ContextLoaderListener将查找一个名为/WEB-INF/applicationContext.xml的文件加载。一旦加载了上下文文件，Spring将基于bean定义创建一个WebApplicationContext对象，并将其存储在Web应用程序的ServletContext中。

所有Java Web框架都构建在Servlet API之上，因此可以使用以下代码片段来访问由ContextLoaderListener创建的“business context”ApplicationContext。

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The `WebApplicationContextUtils` 类是为了方便起见，因此您不必记住ServletContext属性的名称。如果WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE键下不存在一个对象，它的getWebApplicationContext（）方法将返回null。而不是在应用程序中获得NullPointerExceptions的风险，最好使用getRequiredWebApplicationContext（）方法。当ApplicationContext丢失时，此方法抛出异常。

一旦你引用了WebApplicationContext，你可以通过它们的名字或类型检索bean。大多数开发人员通过名称检索bean，然后将其转换为其实现的接口之一.

幸运的是，本节中的大部分框架具有更简单的查找bean的方法。它不仅使得从Spring容器获取bean变得容易，而且还允许您在其控制器上使用依赖注入。每个Web框架部分都有其具体整合策略的更多细节。

21.3 JavaServer Faces 1.2

JavaServer Faces（JSF）是JCP的标准组件，事件驱动的Web用户界面框架。 从Java EE 5开始，它是Java EE umbrella的官方部分.

对于受欢迎的JSF运行时以及流行的JSF组件库，请查看 Apache MyFaces project. MyFaces项目还提供常见的JSF扩展 ，比如MyFaces Orchestra: 一个基于Spring的JSF扩展，可以提供丰富的会话范围支持。

> Spring Web Flow 2.0通过其新建立的Spring Faces模块提供丰富的JSF支持，无论是以JSF为中心的用途（如本节所述）和Spring中心使用（在Spring MVC调度程序中使用JSF视图）。 查看 Spring Web Flow website 了解详情！

Spring的JSF集成中的关键要素是JSF ELResolver机制。

21.3.1 SpringBeanFacesELResolver (JSF 1.2+)

`SpringBeanFacesELResolver` 是符合JSF 1.2的ELResolver实现, 与JSF 1.2和JSP 2.1使用的标准Unified EL集成像SpringBeanVariableResolver一样，它首先将Spring的“business context”WebApplicationContext委托给基于JSF实现的默认解析器.

在配置方面，只需在JSF 1.2 faces-context.xml文件中定义SpringBeanFacesELResolver：

```
<faces-config>
    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
        ...
    </application>
</faces-config>
```

21.3.2 FacesContextUtils

当将属性映射到faces-config.xml中的bean时，自定义VariableResolver可以正常工作，但有时可能需要明确地获取一个bean，The `FacesContextUtils` 类使这个变得容易. 它类似于WebApplicationContextUtils，除了它需要一个FacesContext参数，而不是一个ServletContext参数.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

21.4 Apache Struts 2.x

Craig McClanahan创建,Struts是由Apache Software Foundation托管的一个开源项目。 当时，它大大简化了JSP / Servlet编程范例，并赢得了许多正在使用专有框架的开发人员。 它简化了编程模式，它是开放源代码（因此像啤酒一样自由），它拥有一个大型社区，这使得该项目在Java Web开发人员中成长和流行.

查看Strut Spring Plugin ，用于Struts附带的内置Spring集成。

21.5 Tapestry 5.x

查看 Tapestry homepage:

Tapestry是一个"面向组件的框架，用于在Java中创建动态，强大，高度可扩展的Web应用程序"。

虽然Spring拥有自己强大的 powerful web layer, 但是通过组合Tapestry用于Web用户界面和用于较低层的Spring容器来构建企业Java应用程序，有许多独特的优势。

有关更多信息，请查看 integration module for Spring.

21.6 Further Resources

关于这章节各种Web框架的详解介绍，请点击下面的链接.

- The JSF homepage
- The Struts homepage
- The Tapestry homepage

22. WebSocket 支持

参考文档的这一部分涵盖了Spring框架对Web应用程序中 `WebSocket` 风格消息传递的支持，包括使用STOMP作为应用程序级 `WebSocket` 子协议。

Section 22.1, “Introduction” 建立一个 `WebSocket` 的大致框架，涵盖应用挑战，设计考虑以及何时适合的想法。

Section 22.2,“WebSocket API” 介绍了服务端的 `Spring WebSocket API` ,Section 22.3,“SockJS Fallback Options” 介绍了SockJS 协议，并且展示如何配置和使用它.

Section 22.4.1, “Overview of STOMP” 介绍 STOMP 信息协议. Section 22.4.2, “Enable STOMP over WebSocket” 展示如何在Spring配置STOMP. Section 22.4.4, “Annotation Message Handling” 以下部分说明如何编写注释消息处理方法，发送消息，选择消息代理选项，以及与特殊“用户”目的地的工作. 最后, Section 22.4.18,“Testing Annotated Controller Methods” 列出了测试STOMP / WebSocket 应用程序的三种方法.

22.1 介绍

对于web应用程序，WebSocket 协议RFC 6455定义了一个很重要的功能：全双工，客户端与服务器之间的双向通信. 这是一个令人兴奋的新功能，在漫长的技术历史上，使Web更具交互性，包括Java Applet，XMLHttpRequest，Adobe Flash，ActiveXObject，各种Comet 技术，服务器发送的事件等.

WebSocket协议的介绍超出了本文档的范围。但是，至少要了解，HTTP仅用于初始握手，这依赖于内置于HTTP中的机制来请求协议升级（或在这种情况下为协议交换机），服务器可以使用HTTP状态101对其进行响应 （切换协议）如果它同意。 假设握手成功，HTTP升级请求下面的TCP套接字保持打开，客户端和服务器都可以使用它来彼此发送消息.

Spring Framework 4包括一个全新的WebSocket支持的spring-websocket模块。它与Java WebSocket API标准（JSR-356）兼容，并且还提供额外的附加值，如在其余介绍中所述。

22.1.1 WebSocket 后备选项

采用WebSocket 一个重要的挑战是在一些浏览器中缺乏对其的支持，值得注意的是， IE 第一个支持WebSocket 的版本是10 (详情请参照http://caniuse.com/websockets). 更多,一些限制性代理可以配置为阻止尝试执行HTTP升级或在一段时间后断开连接，因为它已经打开了太久. InfoQ的文章e“How HTML5 Web Sockets Interact With Proxy Servers”中提供了Peter Lubbers对此主题的一个很好的概述。

因此，为了今天构建一个WebSocket应用程序，需要后备选项才能在必要时模拟WebSocket API。 Spring Framework提供了基于SockJS协议的透明后备选项。 这些选项可以通过配置启用，不需要修改应用程序.

22.1.2 消息架构

除了中短期面临的挑战之外，使用WebSocket可以提出重要的设计注意事项，这对于早期的认识至关重要，特别是与我们今天建立Web应用程序相关的知识。

今天REST风格在Web应用中广受欢迎，这是一种依赖于许多URL（资源），少数HTTP方法（动词）以及诸如使用超媒体（链接），以及无状态架构。.

相比之下，WebSocket应用程序可能仅使用单个URL进行初始HTTP握手。此后，所有消息共享并在相同的TCP连接上流动。这指向一个完全不同的，异步的，事件驱动的消息架构。更接近于传统消息传递应用 (如：JMS,AMQP).

Spring Framework 4包括一个新的 spring-messaging 模块 ，其中包含Spring Integration 项目的的关键抽象，例如 Message, MessageChannel, MessageHandler以及其他可以座位消息架构的基础, 该模块还包括一组用于将消息映射到方法的注释，类似于基于Spring MVC注释的编程模型。

22.1.3 WebSocket中的子协议支持

WebSocket确实建立了消息架构，但并不要求使用任何特定的消息协议。它是一个非常窄的TCP层，将字节流转换为消息流（文本或二进制），而不是更多。应用来解释消息的含义。

不同于HTTP（它是应用程序级协议），在WebSocket协议中，框架或容器的传入消息中没有足够的信息来知道如何路由或处理它。因此，WebSocket可以说是太低级别，只是一个非常简单的应用程序。可以做到这一点，但它可能会导致在顶部创

建一个框架。这与目前使用Web框架而不是单独使用Servlet API的大多数Web应用程序相当。

为此，WebSocket RFC定义了子协议的使用。在握手期间，客户端和服务器可以使用头部Sec-WebSocket协议来同意子协议，即较高的应用级协议使用。不需要使用子协议，即使不使用子协议，应用程序仍然需要选择客户端和服务器可以理解的消息格式。该格式可以是自定义，框架特定或标准消息传递协议。

Spring框架支持使用STOMP – 一种简单的消息传递协议，最初创建用于脚本语言，并由HTTP启发的框架。 STOMP被广泛支持，非常适合在WebSocket和Web上使用。

22.1.4 我应该使用WebSocket?

有关使用WebSocket的所有设计考虑，思考“什么时候使用？”是合理的。

WebSocket最适合在Web应用程序中，客户端和服务器需要以高频率和低延迟交换事件。优选的项目类别包括但不限于在金融，游戏，合作等方面的应用。这种应用对时间延迟非常敏感，并且还需要以高频率交换各种各样的消息。

但是，对于其他应用程序类型，可能并非如此。例如，一个新闻或社交软件显示突发新闻，因为它可用可能是完全可以的简单的轮询一次每隔几分钟。这里的延迟很重要，但是如果新闻需要几分钟的时间就可以接受。

即使在延迟至关重要的情况下，如果消息量相对较低（例如监控网络故障），则长时间轮询的使用应被视为一种相对简单的替代方案，其可靠性可靠，并且在效率方面是可比较的的消息相对较低）。

低延迟和高频率的消息可以使WebSocket协议的使用成为关键。即使在这样的应用程序中，选择仍然是所有客户端 – 服务器通信是否应该通过WebSocket消息完成，而不是使用HTTP和REST。答案将因应用而异;然而，有可能某些功能可以通过WebSocket和REST API来暴露，以便为客户提供替代方案。此外，REST API调用可能需要向通过WebSocket连接的感兴趣的客户端广播消息。

Spring Framework允许 `@Controller` 和 `@RestController` 类具有HTTP请求处理和WebSocket消息处理方法。此外，Spring MVC请求处理方法或任何应用方法可以轻松地向所有感兴趣的WebSocket客户端或特定用户广播消息。

22.2 WebSocket API

The Spring架构提供的WebSocket API 被设计成应用于各类WebSocket 引擎. 当前这个列表包括WebSocket 运行时 ，例如 Tomcat 7.0.47+, Jetty 9.1+, GlassFish 4.1+, WebLogic 12.1.3+, and Undertow 1.0+ (and WildFly 8.0+). 随着更多的WebSocket运行时可用，可能会添加额外的支持。

22.2.1 创建和配置一个 WebSocketHandler

创建WebSocket服务器与实现 `WebSocketHandler` 一样简单，或者更有可能扩展 `TextWebSocketHandler` 或 `BinaryWebSocketHandler` ：

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(WebSocketSession session, Text
Message message) {
        // ...
    }

}
```

有专门的 `WebSocket Java-config` 和 `XML` 命名空间支持将上述WebSocket处理程序映射到特定的URL：

```
import org.springframework.web.socket.config.annotation.EnableWe
bSocket;
import org.springframework.web.socket.config.annotation.WebSocke
tConfigurer;
import org.springframework.web.socket.config.annotation.WebSocke
tHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegist
ry registry) {
        registry.addHandler(myHandler(), "/myHandler");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

等价的XML配置:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/webso
cket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-w
ebsocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"
/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHa
ndler"/>

</beans>
```

以上是用于Spring MVC应用程序，应该包含在DispatcherServlet 的配置中.但是Spring的WebSocket支持不依赖于Spring MVC。 在WebSocketHttpRequestHandler的帮助下 ，将WebSocketHttpRequestHandler集成到其他HTTP服务环境中相对简单.

22.2.2 自定义he WebSocket 握手

自定义初始HTTP WebSocket握手请求的最简单的方法是通过 `HandshakeInterceptor` ，它将握手方法之前的“before”和“after”。 这样的拦截器可以用于阻止握手或使任何属性可用于 `WebSocketSession` 。

例如，有一个内置拦截器将HTTP会话属性传递给WebSocket会话：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegist
ry registry) {
        registry.addHandler(new MyHandler(), "/myHandler")
            .addInterceptors(new HttpSessionHandshakeInterceptor
());
    }

}
```

XML等价配置:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/webso
cket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-w
ebsocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"
/>
        <websocket:handshake-interceptors>
            <bean class="org.springframework.web.socket.server.s
upport.HttpSessionHandshakeInterceptor"/>
        </websocket:handshake-interceptors>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHa
ndler"/>

</beans>
```

更高级的选项是扩展执行WebSocket握手步骤的 `DefaultHandshakeHandler` ，包括验证客户端，协商子协议等。 如果需要配置自定义 `RequestUpgradeStrategy` 以适应不支持的WebSocket服务器引擎和版本，应用程序也可能需要使用此选项（有关此主题的更多信息，请参见第22.2.4节“部署注意事项”））。 Java-config和XML命名空间都可以配置自定义HandshakeHandler。

22.2.3 WebSocketHandler 装饰

Spring提供了一个 `WebSocketHandlerDecorator` 基类，可用于使用附加行为来装饰 `WebSocketHandler` 。

在使用WebSocket Java-config或XML命名空间时，默认情况下提供并添加了日志记录和异常处理实现。

`ExceptionWebSocketHandlerDecorator` 捕获从任何 `WebSocketHandler` 方法引发的所有未捕获的异常，并关闭表示服务器错误的状态1011的WebSocket会话。

22.2.4 部署注意事项

Spring WebSocket API易于集成到Spring MVC应用程序中，其中 `DispatcherServlet` 用于HTTP WebSocket握手以及其他HTTP请求。通过调用 `WebSocketHttpRequestHandler` 也很容易集成到其他HTTP处理场景中。这是方便和容易理解。但是，JSR-356运行时可以考虑特殊的考虑因素。

Java WebSocket API（JSR-356）提供了两个部署机制。第一个涉及启动时的Servlet容器类路径扫描（Servlet 3功能）;另一个是在Servlet容器初始化时使用的注册API。这些机制都不可能对所有HTTP处理（包括WebSocket握手和所有其他HTTP请求）使用单个“前端控制器”，例如Spring MVC的 `DispatcherServlet` 。

即使在JSR-356运行时运行时，通过提供特定于服务器的 `RequestUpgradeStrategy` ，Spring的WebSocket支持的JSR-356的一个重大限制。

第二个考虑因素是具有JSR-356支持的Servlet容器预计将执行 `ServletContainerInitializer（SCI）` 扫描，这可能会减慢应用程序启动速度，在某些情况下会显着降低。

如果在升级到支持 `JSR-356` 的Servlet容器版本之后观察到重大影响，则可以通过使用Web.XML中的元素来选择性地启用或禁用Web片段（和SCI扫描）：

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering/>

</web-app>
```

然后，您可以通过名称有选择地启用Web片段，例如Spring自己的 `SpringServletContainerInitializer`，如果需要，可以提供对Servlet 3 Java初始化API的支持:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <absolute-ordering>
        <name>spring_web</name>
    </absolute-ordering>

</web-app>
```

22.2.5 配置WebSocket 引擎

每个底层WebSocket引擎都会公开控制运行时特性的配置属性，例如消息缓冲区大小，空闲超时等。

对于Tomcat，WildFly和GlassFish，在您的WebSocket Java配置中添加一个 `ServletServerContainerFactoryBean` ：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Bean
    public ServletServerContainerFactoryBean createWebSocketCont
ainer() {
        ServletServerContainerFactoryBean container = new Servle
tServerContainerFactoryBean();
        container.setMaxTextMessageBufferSize(8192);
        container.setMaxBinaryMessageBufferSize(8192);
        return container;
    }

}
```

或者WebSocket XML 命名空间:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/webso
cket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-w
ebsocket.xsd">

    <bean class="org.springframework...ServletServerContainerFac
toryBean">
        <property name="maxTextMessageBufferSize" value="8192"/>
        <property name="maxBinaryMessageBufferSize" value="8192"
/>
    </bean>

</beans>
```

对于Jetty, 你需要通过WebSocket Java config预先配置 `Jetty WebSocketServerFactory` 和插件注入到 `Spring’s DefaultHandshakeHandler`

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegist
ry registry) {
        registry.addHandler(echoWebSocketHandler(),
            "/echo").setHandshakeHandler(handshakeHandler());
    }

    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBe
havior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketSer
verFactory(policy)));
    }

}
```

或者WebSocket XML 命名空间:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/webso
cket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
```

```
.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-w
ebsocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/echo" handler="echoHandler"/>
        <websocket:handshake-handler ref="handshakeHandler"/>
    </websocket:handlers>

    <bean id="handshakeHandler" class="org.springframework...Def
aultHandshakeHandler">
        <constructor-arg ref="upgradeStrategy"/>
    </bean>

    <bean id="upgradeStrategy" class="org.springframework...Jett
yRequestUpgradeStrategy">
        <constructor-arg ref="serverFactory"/>
    </bean>

    <bean id="serverFactory" class="org.eclipse.jetty...WebSocke
tServerFactory">
        <constructor-arg>
            <bean class="org.eclipse.jetty...WebSocketPolicy">
                <constructor-arg value="SERVER"/>
                <property name="inputBufferSize" value="8092"/>
                <property name="idleTimeout" value="600000"/>
            </bean>
        </constructor-arg>
    </bean>

</beans>
```

22.2.6允许域名配置

作为 Spring Framework 4.1.5, the default behavior for WebSocket 和SockJS默认的行为仅仅是接收相同的域名请求 . 也可以允许所有或指定的域名列表。 此检查主要是为浏览器客户端设计的。

没有什么可以阻止其他类型的客户端修改Origin标头值(需要查看更多详情：RFC 6454: The Web Origin Concept).

三种可能的行为:

- 只允许相同的域名请求（默认）：在此模式下，当启用SockJS时，Iframe HTTP响应头X-Frame-Options设置为SAMEORIGIN，并且禁用JSONP传输，因为它不允许检查请求的来源。 因此，当启用此模式时，不支持IE6和IE7。
- 允许指定的域名列表：每个提供的允许来源必须以http：//或https：//开头。 在此模式下，当启用SockJS时，基于IFrame和JSONP的传输均被禁用。因此，启用此模式时，不支持IE6至IE9。
- 允许所有域名：启用此模式，您应该提供*作为允许的原始值。 在这种模式下，所有的运输都可用 WebSocket 和SockJS 允许的域名能够如下配置:

```
import org.springframework.web.socket.config.annotation.EnableWe
bSocket;
import org.springframework.web.socket.config.annotation.WebSocke
tConfigurer;
import org.springframework.web.socket.config.annotation.WebSocke
tHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegist
ry registry) {
        registry.addHandler(myHandler(), "/myHandler").setAllowe
dOrigins("http://mydomain.com");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

等价的XML 配置:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/webso
cket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans
.xsd
        http://www.springframework.org/schema/websocket
        http://www.springframework.org/schema/websocket/spring-w
ebsocket.xsd">

    <websocket:handlers allowed-origins="http://mydomain.com">
        <websocket:mapping path="/myHandler" handler="myHandler"
 />
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHa
ndler"/>

</beans>
```

22.3 SockJS后备选项

正如简介所说，As explained in the introduction,WebSocket is not supported in all browsers yet and may be precluded by restrictive network proxies. This is why Spring provides fallback options that emulate the WebSocket API as close as possible based on the SockJS protocol (version 0.3.3).

24. 使用Spring提供远程和WEB服务

24.1 介绍

Spring提供了使用多种技术实现远程访问支持的集成类。远程访问支持使得具有远程访问功能的服务开发变得相当简单，而这些服务由普通的 (Spring) POJO实现。目前，Spring支持以下几种远程技术：

- 远程方法调用（RMI）。通过使用RmiProxyFactoryBean和RmiServiceExporter，Spring同时支持传统的RMI（与java.rmi.Remote接口和java.rmi.RemoteException配合使用）和通过RMI调用器的透明远程调用（透明远程调用可以使用任何Java接口）。
- Spring的HTTP调用器。Spring提供了一个特殊的远程处理策略，允许通过HTTP进行Java序列化，支持任何Java接口（就像RMI调用器）。相应的支持类是HttpInvokerProxyFactoryBean和HttpInvokerServiceExporter。
- Hessian。通过HessianProxyFactoryBean和HessianServiceExporter，可以使用Caucho提供的基于HTTP的轻量级二进制协议来透明地暴露服务。
- JAX-WS。Spring通过JAX-WS为web服务提供远程访问支持。（JAX-WS: 从Java EE 5 和 Java 6开始引入，作为JAX-RPC的继承者)
- JMS。通过JmsInvokerServiceExporter和JmsInvokerProxyFacotryBean类，使用JMS作为底层协议来提供远程服务。
- AMQP。Spring AMQP项目支持AMQP作为底层协议来提供远程服务。

在讨论Spring的远程服务功能时，我们将使用以下的域模型和对应的服务：

```
public class Account implements Serializable{
    private String name;

    public String getName(){
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}

public interface AccountService {

    public void insertAccount(Account account);

    public List<Account> getAccounts(String name);

}

// the implementation doing nothing at the moment
public class AccountServiceImpl implements AccountService {

    public void insertAccount(Account acc) {
        // do something...
    }

    public List<Account> getAccounts(String name) {
        // do something...
    }

}
```

我们将从使用RMI把服务暴露给远程客户端开始，同时讨论使用RMI的一些缺点。然后我们将继续演示一个使用Hessian的例子。

24.2 使用RMI暴露服务

使用Spring的RMI支持，你可以通过RMI基础架构透明地暴露你的服务。完成Spring的RMI设置后，你基本上具有类似于远程EJB配 置，除了没有对安全上下文传递和远程事务传递的标准支持。当使用RMI调用器时，Spring对这些额外的调用上下文提供了钩子，你可以在此插入安全框架 或者自定义的安全凭证。

24.2.1 使用RmiServiceExporter导出服务

使用RmiServiceExporter，我们可以把AccountService对象的接口暴露成RMI对象。可以使用RmiProxyFactoryBean或者在传统RMI服务中使用普通RMI来访问该接口。RmiServiceExporter明确支持使用RMI调用器暴露任何非RMI的服务。

当然，我们首先需要在Spring容器中设置我们的服务：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

下一步我们需要使用RmiServiceExporter来暴露我们的服务：

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter
">
    <!-- does not necessarily have to be the same name as the be
an to be exported -->
    <property name="serviceName" value="AccountService"/>
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

正如你所见，我们覆盖了RMI注册的端口号。通常你的应用服务器还维护一个RMI注册表，明智的做法是不要和它冲突。此外，服务名是用来绑定服务的。现在服务绑定在‘rmi://HOST:1199/AccountService’。我们将在客户端使用这个URL来链接到

服务。

> Note:servicePort属性被省略了(默认值为0).这表示在与服务通信时将使用匿名端口.

24.2.2 在客户端链接服务

我们的客户端是一个使用AccountService来管理account的简单对象：

```
public class SimpleObject {

    private AccountService accountService;

    public void setAccountService(AccountService accountService)
 {
        this.accountService = accountService;
    }

    // additional methods using the accountService

}
```

为了把服务链接到客户端上，我们将创建一个单独的Spring容器，包含这个简单对象和链接配置位的服务：

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这就是我们为支持远程account服务在客户端所需要做的。Spring将透明地创建一个调用器并且通过RmiServiceExporter使得account服务支持远程服务。在客户端，我们用RmiProxyFactoryBean连接它。

24.3 使用Hessian通过HTTP远程调用服务

Hessian提供一种基于HTTP的二进制远程协议。它由Caucho开发的，可以在 http://www.caucho.com 找到更多有关Hessian的信息。

24.3.1 为Hessian和co.配置DispatcherServlet

Hessian使用一个自定义Servlet通过HTTP进行通讯。使用Spring的DispatcherServlet原理，从Spring Web MVC使用中可以看出，可以很容易的配置这样一个Servlet来暴露你的服务。首先我们要在你的应用里创建一个新的Servlet（以下摘录自web.xml）：

```
<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

你可能对Spring的DispatcherServlet很熟悉，这样你将需要在'WEB-INF'目录中创建一个名为'remoting-servlet.xml'(在你的servlet名称后) 的Spring容器配置上下文。这个应用上下文将在下一节中里使用。

或者，可以考虑使用Spring中更简单的HttpRequestHandlerServlet。这允许你在根应用上下文（默认是'WEB-INF/applicationContext.xml'）中嵌入远程exporter定义。每个servlet定义指向特定的exporter bean。在这种情况下，每个servlet的名称需要和目标exporter bean的名称相匹配。

24.3.2 使用HessianServiceExporter暴露你的bean

在新创建的remoting-servlet.xml应用上下文里，我们将创建一个HessianServiceExporter来暴露你的服务：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class="org.springframework.remoting
.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

现在我们准备好在客户端连接服务了。不必显示指定处理器的映射，所以使用BeanNameUrlHandlerMapping把URL请求映射到服务上：因此，服务将通过其包含的bean名称指定的URL导出 DispatcherServlet’s mapping (as defined above): ’http://HOST:8080/remoting/AccountService’ 或者, 在你的根应用上下文中创建一个HessianServiceExporter(比如在’WEB-INF/applicationContext.xml’中):

```
<bean name="accountExporter" class="org.springframework.remoting
.caucho.HessianServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

在后一情况下, 在’web.xml’中为这个导出器定义一个相应的servlet，也能得到同样的结果：这个导出器映射到request路径/remoting/AccountService。注意这个servlet名称需要与目标导出器bean的名称相匹配。

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpR
equestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

24.3.3 在客户端上链接服务

使用HessianProxyFactoryBean，我们可以在客户端链接服务。与RMI示例一样也适用相同的原理。我们将创建一个单独的bean工厂或者应用上下文，并指明SimpleObject使用AccountService来管理accounts的以下bean：

```
<bean class="example.SimpleObject">
    <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.ca
ucho.HessianProxyFactoryBean">
    <property name="serviceUrl" value="http://remotehost:8080/re
moting/AccountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

24.3.4 对通过Hessian暴露的服务使用HTTP基本认证

Hessian的优点之一是，我们可以轻松应用HTTP基本身份验证，因为这两种协议都是基于HTTP的。你的正常HTTP 服务器安全机制可以通过使用web.xml安全功能来应用。通常，你不会为每个用户都建立不同的安全证书，而是在Hessian/BurlapProxyFactoryBean级别共享安全证书（类似一个JDBCDataSource）。

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrl
HandlerMapping">
    <property name="interceptors" ref="authorizationInterceptor"
/>
</bean>

<bean id="authorizationInterceptor"
        class="org.springframework.web.servlet.handler.UserRoleA
uthorizationInterceptor">
    <property name="authorizedRoles" value="administrator,operat
or"/>
</bean>
```

这个是我们显式使用了BeanNameUrlHandlerMapping的例子，并设置了一个拦截器，只允许管理员和操作员调用这个应用上下文中提及的bean。

> Note: 当然，这个例子并不表现出灵活的安全架构。有关安全性方面的更多选项，请查看Spring Security项目http://projects.spring.io/spring-security/。

24.4 使用HTTP调用器暴露服务

与使用自身序列化机制的轻量级协议Hessian相反，Spring HTTP调用器使用标准Java序列化机制通过HTTP暴露业务。如果你的参数或返回值是复杂类型，并且不能通过Hessian的序列化机制进行序列化，HTTP调用器就很有优势（请参阅下一节，以便在选择远程处理技术时进行更多考虑）。

在底层，Spring使用JDK提供的标准工具或Commons的HttpComponents来实现HTTP调用。如果你需要更先进和更易用的功能，请使用后者。你可以参考hc.apache.org/httpcomponents-client-ga/ 以获取更多信息。

24.4.1 暴露服务对象

为服务对象设置HTTP调用器基础架构类似于使用Hessian进行相同操作的方式。就象为Hessian支持提供的HessianServiceExporter，Spring的HTTP调用器提供了org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter。 为了在Spring Web MVC的DispatcherServlet中暴露AccountService(之前章节提及过)，需要在调度程序的应用程序上下文中使用以下配置：

```
<bean name="/AccountService" class="org.springframework.remoting
.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

如Hessian章节部分所述，这个导出器定义将通过DispatcherServlet的标准映射工具暴露出来。 或者， 在你的根应用上下文中(比如'WEB-INF/applicationContext.xml')创建一个HttpInvokerServiceExporter:

```
<bean name="accountExporter" class="org.springframework.remoting
.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

此外，在'web.xml'中为该导出器定义相应的servlet ，其中servlet名称与目标导出器的bean名称相匹配：

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpR
equestHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

如果你在一个servlet容器之外运行程序和使用Oracle的Java6, 那么你可以使用内置的HTTP服务器实现。你可以配置SimpleHttpServerFactoryBean和SimpleHttpInvokerServiceExporter在一起，像下面这个例子一样：

```
<bean name="accountExporter"
        class="org.springframework.remoting.httpinvoker.SimpleHt
tpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>

<bean id="httpServer"
        class="org.springframework.remoting.support.SimpleHttpSe
rverFactoryBean">
    <property name="contexts">
        <util:map>
            <entry key="/remoting/AccountService" value-ref="acc
ountExporter"/>
        </util:map>
    </property>
    <property name="port" value="8080" />
</bean>
```

24.4.2 在客户端连接服务

同样，从客户端连接业务与你使用Hessian所做的很相似。使用代理，Spring可以将你的HTTP POST调用请求转换成被暴露服务的URL。

```
<bean id="httpInvokerProxy" class="org.springframework.remoting.
httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl" value="http://remotehost:8080/re
moting/AccountService"/>
    <property name="serviceInterface" value="example.AccountServ
ice"/>
</bean>
```

如前所述，你可以选择要使用的HTTP客户端。默认情况下，HttpInvokerProxy使用JDK的HTTP功能，但你也可以通过设置httpInvokerRequestExecutor属性来使用ApacheHttpComponents客户端：

```
<property name="httpInvokerRequestExecutor">
    <bean class="org.springframework.remoting.httpinvoker.HttpCo
mponentsHttpInvokerRequestExecutor"/>
</property>
```

24.5 Web 服务

Spring提供了对标准Java Web服务API的全面支持：

- 使用JAX-WS暴露Web服务
- 使用JAX-WS访问Web服务

除了在Spring Core中支持 JAX-WS，Spring portfolio也提供了一种特性Spring Web Services，一种为契约优先和文档驱动的web服务所提供的方案，强烈建议用来创建现代化的，面向未来的web服务。

24.5.1使用JAX- WS暴露基于servlet的web服务

Spring为JAX-WS servlet的端点实现提供了一个方便的基类 – SpringBeanAutowiringSupport. 为了暴露我们的AccountService，我们扩展Spring的SpringBeanAutowiringSupport类并在这里实现了我们的业务逻辑，通常委派调用业务层。我们在Spring管理的bean里面简单地使用Spring的@Autowired 注解来表达这样的依赖关系。

```
/**
 * JAX-WS compliant AccountService implementation that simply de
legates
 * to the AccountService implementation in the root web applicat
ion context.
 *
 * This wrapper class is necessary because JAX-WS requires worki
ng with dedicated
 * endpoint classes. If an existing service needs to be exported
, a wrapper that
 * extends SpringBeanAutowiringSupport for simple Spring bean au
towiring (through
 * the @Autowired annotation) is the simplest JAX-WS compliant w
ay.
 *
 * This is the class registered with the server-side JAX-WS impl
ementation.
 * In the case of a Java EE 5 server, this would simply be defin
```

```
ed as a servlet
 * in web.xml, with the server detecting that this is a JAX-WS e
ndpoint and reacting
 * accordingly. The servlet name usually needs to match the spec
ified WS service name.
 *
 * The web service engine manages the lifecycle of instances of
this class.
 * Spring bean references will just be wired in here.
 */
import org.springframework.web.context.support.SpringBeanAutowir
ingSupport;

@WebService(serviceName="AccountService")
public class AccountServiceEndpoint extends SpringBeanAutowiring
Support {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public Account[] getAccounts(String name) {
        return biz.getAccounts(name);
    }

}
```

我们的AccountServletEndpoint需要和Spring在同一个上下文的web应用里运行，以允许访问Spring的功能。为JAX-WS servlet端点部署使用标准规约是Java EE 5 环境下的默认情况。

24.5.2 使用JAX-WS暴露单独web服务

Oracle JDK 1.6附带的内置JAX-WS provider 使用内置的HTTP服务器来暴露web服务。Spring的SimpleJaxWsServiceExporter类检测所有在Spring应用上下文中配置有@WebService注解的bean，然后通过默认的JAX-WS服务器（JDK 1.6 HTTP服务器）导出。

在这种场景下，端点实例将被作为Spring bean来定义和管理。它们将使用JAX-WS引擎来注册，但其生命周期将由Spring应用程序上下文决定。这意味着Spring的显示依赖注入可用于端点实例。当然通过@Autowired来进行注解驱动的注入也会起作用。

```
<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServi
ceExporter">
    <property name="baseAddress" value="http://localhost:8080/"/
>
</bean>

<bean id="accountServiceEndpoint" class="example.AccountServiceE
ndpoint">
    ...
</bean>

...
```

AccountServiceEndpoint可能来自于Spring的SpringBeanAutowiringSupport，也可能不是。因为这里的端点是由Spring完全管理的bean。这意味着端点实现可能像下面这样没有任何父类定义 – 而且Spring的@Autowired配置注解仍然能够使用：

```
@WebService(serviceName="AccountService")
public class AccountServiceEndpoint {

    @Autowired
    private AccountService biz;

    @WebMethod
    public void insertAccount(Account acc) {
        biz.insertAccount(acc);
    }

    @WebMethod
    public List<Account> getAccounts(String name) {
        return biz.getAccounts(name);
    }

}
```

24.5.3 使用JAX-WS RI的Spring支持来暴露服务

Oracle的JAX-WS RI被作为GlassFish项目的一部分来开发,它使用了Spring支持来作为JAX-WS Commons项目的一部分。这允许把JAX-WS端点作为Spring管理的bean来定义。这与前面章节讨论的单独模式类似 – 但这次是在Servlet环境中。注意这在Java EE 5环境中是不可迁移的，建议在没有EE的web应用环境如Tomcat中嵌入JAX-WS RI。 与标准的暴露基于servlet的端点方式不同之处在于端点实例的生命周期将被Spring管理。这里在web.xml将只有一个JAX-WS servlet定义。在标准的Java EE 5风格中(如上所示)，你将对每个服务端点定义一个servlet，每个服务端点都代理到Spring bean (通过使用@Autowired，如上所示)。 关于安装和使用详情请查阅https://jax-ws-commons.dev.java.net/spring/。

24.5.4 使用JAX-WS访问web服务

Spring提供了2个工厂bean来创建JAX-WS web服务代理，它们是LocalJaxWsServiceFactoryBean和JaxWsPortProxyFactoryBean。前一个只能返回一个JAX-WS服务对象来让我们使用。后面的是可以返回我们业务服务接口的代

理实现的完整版本。这个例子中我们使用后者来为AccountService端点再创建一个代理：

```
<bean id="accountWebService" class="org.springframework.remoting
.jaxws.JaxWsPortProxyFactoryBean">
    <property name="serviceInterface" value="example.AccountServ
ice"/>
    <property name="wsdlDocumentUrl" value="http://localhost:888
8/AccountServiceEndpoint?WSDL"/>
    <property name="namespaceUri" value="http://example/"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountServiceEndpointPort"
/>
</bean>
```

serviceInterface是我们客户端将使用的远程业务接口。wsdlDocumentUrl是WSDL文件的URL. Spring需要用它作为启动点来创建JAX-WS服务。namespaceUri对应.wsdl文件中的targetNamespace。serviceName对应.wsdl文件中的服务名。portName对应.wsdl文件中的端口号。 现在我们可以很方便的访问web服务，因为我们有一个可以将它暴露为AccountService接口的bean工厂。我们可以在Spring中这样使用：

```
<bean id="client" class="example.AccountClientImpl">
    ...
    <property name="service" ref="accountWebService"/>
</bean>
```

从客户端代码上我们可以把这个web服务当成一个普通的类进行访问：

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
    }
}
```

> Note: 上面例子被稍微简化了，因为JAX-WS需要端点接口及实现类来使用@WebService,@SOAPBinding等注解。 这意味着你不能简单地使用普通的Java接口和实现来作为JAX-WS端点，你需要首先对它们进行相应的注解。这些需求详情请查阅JAX-WS文档。

24.6 JMS

使用JMS来作为底层的通信协议透明暴露服务也是可能的。Spring框架中对JMS的远程支持也很基础 – 它在同一线程和同一个非事务 Session上发送和接收，这些吞吐量将非常依赖于实现。需要注意的是这些单线程和非事务的约束仅适用于Spring的JMS远程处理支持。请参见 第26章, JMS (Java消息服务)，Spring对基于JMS的消息的丰富支持。 下面的接口可同时用在服务端和客户端。

```
package com.foo;

public interface CheckingAccountService {

    public void cancelAccount(Long accountId);

}
```

对于上面接口的使用在服务的端简单实现如下：

```
package com.foo;

public class SimpleCheckingAccountService implements CheckingAcc
ountService {

    public void cancelAccount(Long accountId) {
        System.out.println("Cancelling account [" + accountId +
"]");
    }

}
```

这个包含JMS设施的bean的配置文件可同时用在客户端和服务端： <?xml version=”1.0″ encoding=”UTF-8″?>

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
        http://www.springframework.org/schema/beans/spring-beans
.xsd">

    <bean id="connectionFactory" class="org.apache.activemq.Acti
veMQConnectionFactory">
        <property name="brokerURL" value="tcp://ep-t43:61616"/>
    </bean>

    <bean id="queue" class="org.apache.activemq.command.ActiveMQ
Queue">
        <constructor-arg value="mmm"/>
    </bean>

</beans>
```

24.6.1 服务端配置

在服务端你只需要使用JmsInvokerServiceExporter来暴露服务对象。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
        http://www.springframework.org/schema/beans/spring-beans
.xsd">

    <bean id="checkingAccountService"
            class="org.springframework.jms.remoting.JmsInvokerSe
rviceExporter">
        <property name="serviceInterface" value="com.foo.Checkin
gAccountService"/>
        <property name="service">
            <bean class="com.foo.SimpleCheckingAccountService"/>
        </property>
    </bean>

    <bean class="org.springframework.jms.listener.SimpleMessageL
istenerContainer">
        <property name="connectionFactory" ref="connectionFactor
y"/>
        <property name="destination" ref="queue"/>
        <property name="concurrentConsumers" value="3"/>
        <property name="messageListener" ref="checkingAccountSer
vice"/>
    </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Server {

    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext(new String[]{"com/foo
/server.xml", "com/foo/jms.xml"});
    }

}
```

24.6.2 客户端配置

客户端只需要创建一个客户端代理来实现上面的接口(CheckingAccountService)。根据后面的bean定义创建的结果对象可以被注入到其它客户端对象中，而这个代理会负责通过JMS将调用转发到服务端。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
        http://www.springframework.org/schema/beans/spring-beans
.xsd">

    <bean id="checkingAccountService"
            class="org.springframework.jms.remoting.JmsInvokerPr
oxyFactoryBean">
        <property name="serviceInterface" value="com.foo.Checkin
gAccountService"/>
        <property name="connectionFactory" ref="connectionFactor
y"/>
        <property name="queue" ref="queue"/>
    </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicati
onContext;

public class Client {

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationCont
ext(
                new String[] {"com/foo/client.xml", "com/foo/jms
.xml"});
        CheckingAccountService service = (CheckingAccountService
) ctx.getBean("checkingAccountService");
        service.cancelAccount(new Long(10));
    }

}
```

24.7 AMQP

有关更多信息，请参考Spring AMQP Reference Document ‘Spring Remoting with AMQP’ section。

24.8 不实现远程接口自动检测

对远程接口不实现自动探测的主要原因是为了避免向远程调用者打开了太多的大门。目标对象有可能实现的是类似InitializingBean或者DisposableBean这样的内部回调接口，而这些是不希望暴露给调用者的。

提供一个所有接口都被目标实现的代理通常和本地情况无关。但是当暴露一个远程服务时，你应该只暴露特定的用于远程使用的服务接口。除了内部回调接口，目标有可能实现了多个业务接口，而往往只有一个是用于远程调用的。出于这些原因，我们要求指定这样的服务接口。

这是在配置方便性和意外暴露内部方法的危险性之间作的权衡。始终指定一个服务接口并不需要花太大代价，并可以令控制具体方法暴露更加安全。

24.9 选择技术时的注意事项

这里提到的每种技术都有它的缺点。你在选择一种技术时，应该仔细考虑你的需要和所暴露的服务及你在远程访问时传送的对象。

当使用RMI时，通过HTTP协议访问对象是不可能的，除非你正在HTTP通道传输RMI流量。RMI是一种重量级协议，因为它支持整个对象的序列化，当要求网络上传输复杂数据结构时这是非常重要的。然而，RMI-JRMP与Java客户端相关：它是一种Java-to-Java的远程访问解决方案。

如果你需要基于HTTP的远程访问而且还要求使用Java序列化，Spring的HTTP调用器是一个不错的选择。它和RMI调用器共享相同的基础设施，只需使用HTTP作为传输。注意HTTP调用器不仅限于Java-to-Java的远程访问，而且还限于使用Spring的客户端和服务器端。（后者也适用于Spring的RMI调用器，用于非RMI接口。）

Hessian可以在异构环境中运行时提供重要的价值，因为它们明确允许非Java客户端。然而，非Java支持仍然有限。已知问题包括将Hibernate对象与延迟初始化的集合相结合的序列化。如果您有这样的数据模型，请考虑使用RMI或HTTP调用者而不是Hessian。

在使用服务集群和需要JMS代理（JMS broker）来处理负载均衡及发现和自动-失败恢复服务时JMS是很有用的。缺省情况下，在使用JMS远程服务时使用Java序列化，但是JMS提供者也可以使用不同的机制例如XStream来让服务器用其他技术。

最后但并非最不重要的是，EJB比RMI具有优势，因为它支持标准的基于角色的身份认证和授权，以及远程事务传递。用RMI调用器或HTTP调用器来支持安全上下文的传递是可能的，虽然这不由核心core Spring提供：Spring提供了合适的钩子来插入第三方或定制的解决方案。

24.10 在客户端访问RESTful服务

RestTemplate是客户端访问RESTful服务的核心类。它在概念上类似于Spring中的其他模板类，例如JdbcTemplate、 JmsTemplate和其他Spring组合项目中发现的其他模板类。

RestTemplate’s behavior is customized by providing callback methods and configuring the `HttpMessageConverter用于将对象打包到HTTP请求体中，并将任何响应解包成一个对象。通常使用XML作为消息格式，Spring提供了MarshallingHttpMessageConverter，它使用了的Object-to-XML框架，也是org.springframework.oxm包的一部分。这为你提供了各种各样的XML到对象映射技术的选择。

本节介绍如何使用RestTemplate它及其关联 的HttpMessageConverters。

24.10.1 RestTemplate

在Java中调用RESTful服务通常使用助手类（如Apache HttpComponents）完成HttpClient。对于常见的REST操作，此方法的级别太低，如下所示。

```
String uri = "http://example.com/hotels/1/bookings";

PostMethod post = new PostMethod(uri);
String request = // create booking request content
post.setRequestEntity(new StringRequestEntity(request));

httpClient.executeMethod(post);

if (HttpStatus.SC_CREATED == post.getStatusCode()) {
    Header location = post.getRequestHeader("Location");
    if (location != null) {
        System.out.println("Created new booking at :" + location
.getValue());
    }
}
```

RestTemplate提供了更高级别的方法，这些方法与六种主要的HTTP方法中的每一种相对应，这些方法使得调用许多RESTful服务成为一个单行和执行REST的最佳实践。

Note: RestTemplate具有异步计数器部分：请参见第24.10.3节“异步RestTemplate”。

Table 24.1. RestTemplate方法概述

HTTP Method	RestTemplate Method
DELETE	delete
GET	getForObject getForEntity
HEAD	headForHeaders(String url, String… uriVariables)
OPTIONS	optionsForAllow(String url, String… uriVariables)
POST	postForLocation(String url, Object request, String… uriVariables) postForObject(String url, Object request, Class<T> responseType, String… uriVariables)
PUT	put(String url, Object request, String…uriVariables)
PATCH and others	exchange execute

RestTemplate方法名称遵循命名约定，第一部分指出正在调用什么HTTP方法，第二部分指出返回的内容。例如，该方法getForObject()将执行GET，将HTTP响应转换为你选择的对象类型并返回该对象。方法postForLocation() 将执行POST，将给定对象转换为HTTP请求，并返回可以找到新创建的对象的响应HTTP Location头。在异常处理HTTP请求的情况下，RestClientException类型的异常将被抛出; 这个行为可以在RestTemplate通过插入另一个ResponseErrorHandler实现来改变。

exchange和execute方法是上面列出的更具体的方法的广义版本，并且可以支持额外的组合和方法，例如HTTP PATCH。但是，请注意，底层HTTP库还必须支持所需的组合。JDK HttpURLConnection不支持该PATCH方法，但Apache HttpComponents HttpClient4.2或更高版本支持。他们还能够通过使用一个能够捕获和传递通用类型信息的新类ParameterizedTypeReference来使得RestTemplate能够读取通用类型的HTTP响应信息（例如List）。

对象通过HttpMessageConverter实例传递给这些方法并从这些方法返回被转换为HTTP消息。主要mime类型的转换器默认注册，但你也可以编写自己的转换器并通过messageConverters()实体属性注册它 。模板默认注册的转换器实例是ByteArrayHttpMessageConverter，StringHttpMessageConverter，FormHttpMessageConverter和SourceHttpMessageConverter。如果使用MarshallingHttpMessageConverter或者MappingJackson2HttpMessageConverter，你可以使用messageConverters()实体属性覆盖这些默认值。

每个方法以两种形式使用URI模板参数，作为String可变长度参数或Map<String,String>。例如，使用可变长参数如下：

```
String result = restTemplate.getForObject(
    "http://example.com/hotels/{hotel}/bookings/{booking}", Stri
ng.class,"42", "21");
```

使用一个Map<String,String>如下：

```
Map<String, String> vars = Collections.singletonMap("hotel", "42
");
String result = restTemplate.getForObject(
        "http://example.com/hotels/{hotel}/rooms/{hotel}", Strin
g.class, vars);
```

要创建一个实例，RestTemplate可以简单地调用默认的无参数构造函数。这将使用java.net包中的标准Java类作为底层实现来创建HTTP请求。这可以通过指定实现来覆盖ClientHttpRequestFactory。Spring提供了HttpComponentsClientHttpRequestFactory使用Apache HttpComponents HttpClient创建请求的实现。HttpComponentsClientHttpRequestFactory通过使用一个可以配置凭证信息或连接池功能的org.apache.http.client.HttpClient实例来配置。

> Note: HTTP请求的java.net实现可能会在访问表示错误的响应状态（例如401）时引发异常。如果这是一个问题，请切换到HttpComponentsClientHttpRequestFactory。

前面使用Apache HttpCOmponentsHttpClientdirectly的例子用RestTemplate重写如下：

```
uri = "http://example.com/hotels/{id}/bookings";

RestTemplate template = new RestTemplate();

Booking booking = // create booking object

URI location = template.postForLocation(uri, booking, "1");
```

使用Apache HttpComponents, 而不是原生的java.net功能，构造RestTemplate如下：

```
RestTemplate template = new RestTemplate(new HttpComponentsClien
tHttpRequestFactory());
```

> Note: Apache HttpClient 支持gzip编码，要使用这个功能，构造HttpCOmponentsClientHttpRequestFactory如下：
>
> ```
> HttpClient httpClient = HttpClientBuilder.create().build();
> ClientHttpRequestFactory requestFactory = new HttpComponents
> ClientHttpRequestFactory(httpClient);
> RestTemplate restTemplate = new RestTemplate(requestFactory)
> ;
> ```

当execute方法被调用，通用的回调接口是RequestCallback并且会被调用。

```
public <T> T execute(String url, HttpMethod method, RequestCallb
ack requestCallback,
        ResponseExtractor<T> responseExtractor, String... uriVar
iables)

// also has an overload with uriVariables as a Map<String, Strin
g>.
```

RequestCallback接口定义如下：

```
public interface RequestCallback {
 void doWithRequest(ClientHttpRequest request) throws IOExceptio
n;
}
```

允许您操作请求标头并写入请求主体。当使用execute方法时，你不必担心任何资源管理，模板将始终关闭请求并处理任何错误。有关使用execute方法及它的其他方法参数的含义的更多信息，请参阅API文档。

使用URI

对于每个主要的HTTP方法，RestTemplate提供的变体使用String URI或java.net.URI作为第一个参数。
String URI变体将模板参数视为String变长参数或者一个Map<String,String>。他们还假定URL字符串不被编码且需要编码。例如：

```
restTemplate.getForObject("http://example.com/hotel list", Strin
g.class);
```

将在 http://example.com/hotel list执行一个GET请求。这意味着如果输入的URL字符串已被编码，它将被编码两次 – 即将 http://example.com/hotel list变为 http://example.com/hotel list。如果这不是预期的效果，则使用java.net.URI方法变体，假设URL已经被编码，如果要重复使用单个（完全扩展）URI多次，通常也是有用的。

UriComponentsBuilder类可用于构建和编码URI包括URI模板的支持。例如，你可以从URL字符串开始：

```
UriComponents uriComponents = UriComponentsBuilder.fromUriString
( "http://example.com/hotels/{hotel}/bookings/{booking}").build(
)
        .expand("42", "21")
        .encode();

URI uri = uriComponents.toUri();
```

或者分别制定每个URI组件:

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
        .scheme("http").host("example.com").path("/hotels/{hotel
}/bookings/{booking}").build()
        .expand("42", "21")
        .encode();

URI uri = uriComponents.toUri();
```

处理请求和响应头

除了上述方法之外，RestTemplate还具有exchange() 方法，可以用于基于HttpEntity 类的任意HTTP方法执行。
也许最重要的是，该exchange()方法可以用来添加请求头和读响应头。例如：

```
HttpHeaders requestHeaders = new HttpHeaders();
requestHeaders.set("MyRequestHeader", "MyValue");
HttpEntity<?> requestEntity = new HttpEntity(requestHeaders);

HttpEntity<String> response = template.exchange( "http://example
.com/hotels/{hotel}",
        HttpMethod.GET, requestEntity, String.class, "42");

String responseHeader = response.getHeaders().getFirst("MyRespon
seHeader");
String body = response.getBody();
```

在上面的例子，我们首先准备了一个包含MyRequestHeader 头的请求实体。然后我们检索返回和读取MyResponseHeader和消息体。

Jackson JSON 视图支持

可以指定一个 Jackson JSON视图来系列化对象属性的一部分，例如：

```
MappingJacksonValue value = new MappingJacksonValue(new User("er
ic", "7!jd#h23"));
value.setSerializationView(User.WithoutPasswordView.class);
HttpEntity<MappingJacksonValue> entity = new HttpEntity<MappingJ
acksonValue>(value);
String s = template.postForObject("http://example.com/user", ent
ity, String.class);
```

24.10.2 HTTP 消息转换

通过HttpMessageConverters，对象传递到和从getForObject(),postForLocation(),和put()这些方法返回被转换成HTTP请求和HTTP相应。HttpMessageConverter接口如下所示，让你更好地感受它的功能：

```
public interface HttpMessageConverter<T> {

    // Indicate whether the given class and media type can be re
ad by this converter.
    boolean canRead(Class<?> clazz, MediaType mediaType);

    // Indicate whether the given class and media type can be wr
itten by this converter.
    boolean canWrite(Class<?> clazz, MediaType mediaType);

    // Return the list of MediaType objects supported by this co
nverter.
    List<MediaType> getSupportedMediaTypes();

    // Read an object of the given type from the given input mes
sage, and returns it.
    T read(Class<T> clazz, HttpInputMessage inputMessage) throws
 IOException, HttpMessageNotReadableException;

    // Write an given object to the given output message.
    void write(T t, HttpOutputMessage outputMessage) throws IOEx
ception, HttpMessageNotWritableException;

}
```

框架中提供主要媒体（mime）类型的具体实现，默认情况下，通过RestTemplate在客户端和 RequestMethodHandlerAdapter在服务器端注册。

HttpMessageConverter的实现下面章节中描述。对于所有转换器，使用默认媒体类型，但可以通过设置supportedMediaTypesbean属性来覆盖。

StringHttpMessageConverter

一个HttpMessageConverter的实现，实现从HTTP请求和相应中读和写Strings。默认情况下，该转换器支持所有的文本媒体类型（text/*）,并用text/plain的Content-Type来写。

FormHttpMessageConverter

一个HttpMessageConverter的实现，实现从HTTP请求和响应读写表单数据。默认情况下，该转换器读写application/x-www-form-urlencoded媒体类型。表单数据被读取并写入MultiValueMap<String, String>。

ByteArrayHttpMessageConverter

一个HttpMessageConverter的实现，实现从HTTP请求和响应中读取和写入字节数组。默认情况下，此转换器支持所有媒体类型（/），并使用其中的一种Content-Type进行写入application/octet-stream。这可以通过设置supportedMediaTypes属性和覆盖getContentType(byte[])来重写。

MarshallingHttpMessageConverter

一个HttpMessageConverter的实现，从org.springframework.oxm包中使用Spring的Marshaller和Unmarshaller抽象实现读取和写入XML。该转换器需要Marshaller和Unmarshaller才能使用它。这些可以通过构造函数或bean属性注入。默认情况下，此转换器支持（ text/xml）和（application/xml）。

MappingJackson2HttpMessageConverter

一个HttpMessageConverter的实现，使用Jackson XML扩展的ObjectMapper实现读写JSON。可以根据需要通过使用JAXB或Jackson提供的注释来定制XML映射。当需要进一步控制时，XmlMapper 可以通过ObjectMapper属性注入自定义，以便需要为特定类型提供自定义XML序列化器/反序列化器。默认情况下，此转换器支持（application/xml）。

MappingJackson2XmlHttpMessageConverter

一个HttpMessageConverter的实现，可以使用Jackson XML扩展的XmlMapper读取和写入XML。可以根据需要通过使用JAXB或Jackson提供的注释来定制XML映射。当需要进一步控制时，XmlMapper 可以通过ObjectMapper属性注入自定义，以便需要为特定类型提供自定义XML序列化器/反序列化器。默认情况下，此转换器支持（application/xml）。

SourceHttpMessageConverter

一个HttpMessageConverter的实现，从HTTP请求和响应中读写javax.xml.transform.Source。仅支持DOMSource、SAXSource和StreamSource。默认情况下，此转换器支持（text/xml）和（application/xml）。

BufferedImageHttpMessageConverter

一个HttpMessageConverter的实现，从HTTP请求和响应中读写java.awt.image.BufferedImage。此转换器读写Java I/O API支持的媒体类型。

24.10.3 异步RestTemplate

Web应用程序通常需要查询外部REST服务。当为这些需求扩张应用程序时，HTTP和同步调用的性质带来挑战：可能会阻塞多个线程，等待远程HTTP响应。

AsyncRestTemplate和第24.10.1节“RestTemplate”的API非常相似; 请 参见表24.1“RestTemplate方法概述”。这些API之间的主要区别是AsyncRestTemplate返回ListenableFuture 封装器而不是具体的结果。

前面的RestTemplate例子翻译成：

```
// async call
Future<ResponseEntity<String>> futureEntity = template.getForEntity(
    "http://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21");

// get the concrete result - synchronous call
ResponseEntity<String> entity = futureEntity.get();
```

ListenableFuture 接受完成回调：

```
ListenableFuture<ResponseEntity<String>> futureEntity = template
.getForEntity(
    "http://example.com/hotels/{hotel}/bookings/{booking}", Stri
ng.class, "42", "21");

// register a callback
futureEntity.addCallback(new ListenableFutureCallback<ResponseEn
tity<String>>() {
    @Override
    public void onSuccess(ResponseEntity<String> entity) {
        //...
    }

    @Override
    public void onFailure(Throwable t) {
        //...
    }
});
```

> Note: 默认AsyncRestTemplate构造函数为执行HTTP请求注册一个SimpleAsyncTaskExecutor 。当处理大量短命令请求时，线程池的TaskExecutor实现ThreadPoolTaskExecutor 可能是一个不错的选择。

有关更多详细信息，参考ListenableFuture的javadoc and AsyncTestTmeplate的javadoc.

25. 整合EJB

25.1 介绍

作为一个轻量级的容器，Spring通常被认为是EJB的替代品。我们相信对域大多数就算不是最多的应用和用例来说，Spring作为一个容器结合其丰富的在事物，ORM和JDBC访问方面的支持功能，是比通过一个EJB容器和EJBs来实现同等的功能更好的选择。

然后，需要注意的是使用Spring并不会阻止你使用EJBs。实际上，Spring使访问EJBs，实现EJBs以及其中的功能更加容易。另外的，使用Spring来获取EJBs提供的服务可以允许这些服务透明的在本地EJB，远程EJB或者POJO(Plain Old Java Object)变量中切换，而不需要客户端代码做改变。

在这一章，我们会看到Spring使如何帮助你获取和实现EJBs。Spring提供特定的值在访问无状态会话组件(SLSBs)，所以我们从这里开始讨论。

25.2 获取EJBs

25.2.1 概念

调用一个本地或者远程的无状态会话组件中的方法，客户端代码必须正常执行JNDI查找来获取本地或者远程的EJB Home Object，然后使用'create' 方法来获取一个真正的（本地或者远程的）EJB对象。然后一个或者多个方法会在EJB中被调用。为了避免重复底层级的代码，很多EJB应用使用服务定位器和业务代表模式。它们比起在客户端代码中到处使用JNDI lookup要好，但是也有很明显不好的地方，比如：

- 通常使用EJBs的代码依赖于服务定位器或者业务代理单例，所以很难测试。
- 在使用了服务定位器模式而不是用一个业务代理的情况下，应用代码仍然需要结束时在一个EJB home上调用create()方法，并且处理结果中的异常。所以它仍然和EJB的API绑在一起，还会有EJB编程模型的复杂性。
- 实现业务代理模式常常会带来明显的代码重复，我们不得不写很多的方法只是简单的调用EJB中相同的方法。

25.2.2 访问本地无状态会话组件(SLSBs)

假设我们有一个web容器需要使用到本地的EJB。我们将会遵循最好的实践并且使用EJB业务方法接口模式，所以就是EJB的本地接口扩展一个不是EJB-specific的业务方法接口。让我们叫这个接口为 `MyComponent` 。

```
public interface MyComponent {
    ...
}
```

一个主要使用业务方法接口模式的原因是为了确保本地接口中的方法签名和bean实现类同步是自动的。另外的一个原因是它让我们可以在需要的时候更加容易转换为POJO来实现这个服务。当然我们也会需要去实现本地的home接口并且提供一个实现类来实现 `SeesionBean` 和 `MyComponent` 业务方法接口。现在我们唯一需要的Java编码是暴露一个控制器中setter方法进行设置 `MyComponent` 从而将我们的web层控制器连接到这个EJB实现。这将会在容器中保存引用作为一个实例变量：

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

我们之后可以使用这个实例变量在这个控制器中的任何一个业务方法中。现在假设我们在一个Spring控制器以外获取了我们的控制器对象，我们可以在(相同上下文中)配置一个 `LocalStatelessSessionProxyFactoryBean` 实例，这是一个EJB代理对象。这个代理的配置以及控制器的 `myComponet` 属性值是在一个配置项完成的，如下：

```
<bean id="myComponent"
        class="org.springframework.ejb.access.LocalStatelessSess
ionProxyFactoryBean">
    <property name="jndiName" value="ejb/myBean"/>
    <property name="businessInterface" value="com.mycom.MyCompon
ent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

这个配置的生效背后Spring AOP框架做了很多的工作，尽管你并不强制性需要使用AOP观念就能享受到这个结果了。这个 `myComponet` bean定义了一个EJB的代理实现了业务接口。这个EJB local home在启动的时候会被缓存，所以这只有一个单独的JNDI lookup。每当这个EJB被调用的时候，这个代理就会调用本地EJB的 `classname` 方法并且调用EJB上相应的业务方法。
`myController` bean定义为这个EJB代理设置了控制类的 `myComponet` 的属性。
可选的(最好是在许多这样代理定义的情况下)，考虑使用 `<jee:local-slsb>` 在Spring"jee"命名空间中配置元素：

```
<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"
        business-interface="com.mycom.MyComponent"/>

<bean id="myController" class="com.mycom.myController">
    <property name="myComponent" ref="myComponent"/>
</bean>
```

这种EJB获取机制极大的简化了应用代码：web层的代码(或者其他EJB客户端代码)对使用EJB没有依赖。如果我们想要使用一个POJO或者一个模拟对象获取其他的测试桩来替代这个EJB的引用，我们可以简单的修改这个 `myComponet` bean的定义而不需要修改任何Java代码。另外的，我们也不需要在我们应用中写任何一行JNDI lookup或者其他EJB样板代码。
Benchmarks和其他在实际应用中的经验表明这种方法(涉及目标EJB的反射调用)的性能开销是很小的，通常在使用中是不可检测到的。记住我们并不想要细密度的导出调用EJBs，因为在应用服务器中有使用EJB基础架构的开销。

这里有一个关于JNDI lookup的警告。在一个bean容器中，类最好作为一个单例来使用(似乎没有道理把它作为原型使用).然而，如果那个bean容器预先实例化了单例你也许会遇到问题如果这个Bean容器在EJB容器加载目标EJB之前加载了。因为这个JNDI lookup会在这个类的init()方法中调用并且被缓存，但是这个EJB还没有绑定到对应的目标地址。这个解决的方法是不要预先实例化这个工程对象，而是在第一次使用的时候再创建它。在XML容器中，它被 `lazy-init` 属性控制。

尽管这不会是大多是大多数Spring用户的兴趣所在，那些使用EJB做AOP编程的用户也许想要看看 `LocalSlsbInvokerInterceptor`

25.2.3 获取远程的无状态会话组件(SLSBs)

获取远程EJBs基本上和获取本地EJBs相同，除了 `SimpleRemoteStatelessSessionProxyFactoryBean` 或者 `<jee:remote-slsb>` 配置元素会被用到。当然不论用不用到Spring，远程调用语义应用；一个调用在其他VM其他电脑上的对象的方法有时需要被不同的对待根据应用场景和失败的句柄。

Spring的EJB客户端再一次比非Spring方法更加有优势。通常EJB客户端代码要轻松的在调用EJBs本地和远程之间切换是有问题的。因为远程的接口方法必须声明它们可能抛出 `RemoteException` ，而客户端代码必须处理它，而本地的接口没有这个异常。客户端处理本地EJBs的代码如果要改为处理远程EJBs通常需要修改增加处理远程异常，并且客户端如果本来是处理远程EJBs的需要改为处理本地EJBs的代码可以保持不变但是做了很多不必要的处理远程异常的事情，或者就需要移除这部分代码。使用Spring 远程EJB代理你可以不用声明任何抛出 `RemoteException` 的业务方法接口和实现EJB代码，这里有一个远程接口除了抛出异常以外完全相同，而是依靠代理来动态的处理这两个接口使它们表现的一样。这样客户端代码无需处理 `RemoteException` 类。任何在调用这个EJB过程中抛出的 `RemoteException` 将会被再次抛出作为没有没检查的 `RemoteAccessException` 类，这个类是 `RuntimeException` 的子类。这个目标服务可以在不需要客户端代码知晓的情况下在本地EJB或者远程EJB(或者甚至是简单的Java对象)实现中切换。当然，这个是可选择的；你也可以在你的业务接口中声明 `RemoteException` 。

25.2.4 获取EJB 2.x SLSBs和EJB 3 SLSBs的比较

通过Spring获取EJB 2.x会话组件和EJB3会话组件基本上是透明的。Spring的EJB获取器包含了 `<jee:local-slsb>` 和 `<jee:remote-slsb>` ，会在运行时透明的采用实际的组件。它们如果找到了home interface（EJB 2.x 风格）就会获取，反之就直接运行组件调用(EJB 3风格)。

Note：对于EJB 3会话组件，你也可以直接的使用 `JndiObjectFactoryBean` ／ `jee:jndi-lookup` 因为完全有效的组件引用在这里被暴露给了JNDI loolups。定义显示的 `<jee:local-slsb>` ／ `<jee:remote-slsb>` lookups只是提供了一致和更加显示的EJB访问配置。

26. JMS

26.1 介绍

Spring 提供了一个 JMS 的集成框架，简化了 JMS API 的使用，就像 Spring 对 JDBC API 的集成一样。

JMS 大致可分为两块功能，即消息的生产与消费。 `JmsTemplate` 类用于消息生产和消息的同步接收。 对于类似 Java EE 的消息驱动 Bean 形式的异步接收，Spring 提供了大量用于创建消息驱动 POJOs（MDPs）的消息监听器。Spring 还提供了一种创建消息侦听器的声明式方法。

`org.springframework.jms.core` 包提供了使用 JMS 的核心功能。它包含了 JMS 模板类，用来处理资源的创建与释放，从而简化 JMS 的使用，就像 `JdbcTemplate` 对 JDBC 做的一样。像其他大多数 Spring 模板类一样，JMS 模板类提供了执行公共操作的 helper 方法。在需要更复杂应用的情况下，类把处理任务的核心委托给用户实现的回调接口。JMS 类提供了方便的方法，用来发送消息、同步地使用消息以及向用户公开 JMS 会话和消息的生产者。

`org.springframework.jms.support` 包提供了转换 JMSException 的功能。 转换代码将受检查的 `JMSException` 层转换到不受检查异常的镜像层。如果有一个提供者指定的受检查的 `javax.jms.JMSException` 类的子类，这个子类异常被封装到了不受检查的 `UncategorizedJmsException` 异常中。

`org.springframework.jms.support.converter` 包提供了 MessageConverter 抽象，进行 Java 对象和 JMS 消息的互相转换。

`org.springframework.jms.support.destination` 包提供了管理 JMS 目的地的不同策略，比如针对 JNDI 中保存的目标的服务定位器。

`org.springframework.jms.annotation` 包提供了支持注解驱动监听端点的必要基础架构，通过使用 `@JmsListener` 实现。

`org.springframework.jms.config` 包提供了 JMS 命名空间的解析实现，以及配置监听容器和创建监听端点的 java 配置支持。

最后， `org.springframework.jms.connection` 包提供了适用于独立应用程序的 `ConnectionFactory` 实现。 它还包含 Spring 对 JMS 的 `PlatformTransactionManager` 实现（即 `JmsTransactionManager` ）。这将允许 JMS 作为事务性资源无缝集成到 Spring 的事务管理机制中。

26.2 Spring JMS的使用

26.2.1 JmsTemplate

`JmsTemplate` 类是JMS核心包中的中心类。它简化了 JMS 的使用，因为在发送或同步接收消息时它帮我们处理了资源的创建和释放。

使用 `JmsTemplate` 的代码只需要实现规范中定义的回调接口。在 `JmsTemplate` 中通过调用代码让 `MessageCreator` 回调接口用所提供的会话(`Session`)创建消息。然而，为了顾及更复杂的 JMS API 应用，回调接口 `SessionCallback` 将 JMS 会话提供给用户，回调接口 `ProducerCallback` 则公开了 `Session` 和 `MessageProducer` 的组合。

JMS API 公开了发送方法的两种类型，一种接受交付模式、优先级和存活时间作为服务质量（QOS）参数，另一种则使用缺省值作为 QOS 参数（无需参数）方式。由于 JmsTemplate 中有很多发送方法，QOS 参数用 bean 属性进行暴露设置，从而避免在一系列发送方法中的重复。同样地，使用 `setReceiveTimeout` 属性设置用于同步接收调用的超时值。

一些 JMS 提供者通过配置 `ConnectionFactory` ，管理方式上允许默认的 QOS值的设置。 `MessageProducer` 的发送方法 `send(Destination destination, Message message)` 在那些专有的 JMS 中将会使用不一样的 QOS 默认值。 所以，为了提供对 QOS 值域、的统一管理， `JmsTemplate` 必须通过设置布尔值属性 `isExplicitQosEnabled` 为true，使它能够使用自己的QOS值。

为了方便起见， `JmsTemplate` 还暴露了一个基本的请求-回复操作，允许在一个作为操作一部分而被创建的临时队列上，进行消息的发送与等待回复。

> 配置的 `JmsTemplate` 类的实例是线程安全的。这很重要，因为这意味着你可以配置一个 `JmsTemplate` 单例，然后安全地将这个共享引用注入给多个协作者。 要清楚，保持对 `ConnectionFactory` 引用的 `JmsTemplate` 是有状态的，但该状态不是会话状态。

从 Spring Framework 4.1开始， `JmsMessagingTemplate` 构建在 `JmsTemplate` 之上，并提供与消息抽象层（即 `org.springframework.messaging.Message` ）的集成。 这允许你以通用的方式来创建要发送的消息。

26.2.2 Connections

`JmsTemplate` 需要一个对 `ConnectionFactory` 的引用。`ConnectionFactory` 是 JMS 规范的一部分，并被作为使用 JMS 的入口。客户端应用通常作为一个工厂配合 JMS 提供者去创建连接，并封装一系列的配置参数，其中一些是和供应商相关的，例如 SSL 的配置选项。

当在 EJB 内使用 JMS 时，供应商提供 JMS 接口的实现，以至于可以参与声明式事务的管理，提供连接池和会话池。为了使用这个实现，J2EE 容器一般要求你在 EJB 或 servlet 部署描述符中将 JMS 连接工厂声明为 resource-ref。为确保可以在 EJB 内使用 `JmsTemplate` 的这些特性，客户端应当确保它能引用其中的 `ConnectionFactory` 实现。

缓存消息资源

标准的API涉及创建许多中间对象。要发送消息，将执行以下“API”步骤

```
ConnectionFactory->Connection->Session->MessageProducer->send
```

在 `ConnectionFactory` 和 `Send` 操作之间，有三个中间对象被创建和销毁。 为了优化资源使用并提高性能，提供了两个 `ConnectionFactory` 的实现。

SingleConnectionFactory

Spring 提供 `ConnectionFactory` 接口的一个实现， `SingleConnectionFactory` ，它将在所有的 `createConnection` 调用中返回同一个的连接，并忽略 `close` 的调用。这在测试和独立的环境中相当有用，因为同一个连接可以被用于多个 `JmsTemplate` 调用以跨越多个事务。`SingleConnectionFactory` 接受一个通常来自 JNDI 的标准 `ConnectionFactory` 的引用。

CachingConnectionFactory

`CachingConnectionFactory` 扩展了 `SingleConnectionFactory` 的功能，它添加了会话、消息生产者、消息消费者的缓存。 初始缓存大小设置为1，使用 `sessionCacheSize` 属性来增加缓存会话的数量。请注意，实际缓存会话的数量将超过该值，因为会话的缓存是基于确认模式的，因此当设

置 `sessionCacheSize` 为1时，缓存的会话可能达到4个，因为每个确认模式都会缓存一个。当缓存的时候，消息生产者和消息消费者被缓存在他们自己的会话中同时也考虑到生产者和消费者的唯一属性。消息生产者基于他们的目的地被缓存，消息消费者基于目的地、选择器、非本地传送标识和持久订阅名称（假设创建持久消费者）的组合键被缓存。

26.2.3 Destination 管理

目的地（Destination），像 `ConnectionFactories` 一样，是可以在 JNDI 中进行存储和提取的 JMS 管理对象。当配置一个 Spring 应用上下文，可以使用 JNDI 工厂类 `JndiObjectFactoryBean / <jee:jndi-lookup>` 将你的对象引用依赖注入到 JMS 目的地。然而，如果在应用中有大量的目的地，或者 JMS 供应商提供了特有的高级目的地管理特性，这个策略常常显得很笨重。高级目的地管理的例子如创建动态目的地或支持目的地的命名层次。 `JmsTemplate` 将目的地名称到 JMS 目的地对象的解析委派给一个 `DestinationResolver` 接口的实现。 `DynamicDestinationResolver` 是 `JmsTemplate` 使用的默认实现，并且提供动态目的地解析。同时 `JndiDestinationResolver` 作为 JNDI 包含的目的地的服务定位器，并且可选择地退回来使用 `DynamicDestinationResolver` 提供的行为。

相当常见的是在一个 JMS 应用中所使用的目的地只有在运行时才知道，因此，当一个应用被部署时，它不能被创建。这经常是因为交互系统组件之间的共享应用逻辑是在运行时按照已知的命名规范创建目的地。虽然动态目的地的创建不是 JMS 规范的一部分，但是许多供应商已经提供了这个功能。用户为所建的动态目的地定义名称，这样区别于临时的目的地，并且动态目的地不会被注册到 JNDI 中。创建动态目的地所使用的 API 在不同的供应商之间差别很大，因为目的地所关联的属性是供应商特有的。然而，有时由供应商作出的一个简单的实现选择是忽略 JMS 规范中的警告，并使用 `TopicSession` 的方法 `createTopic(String topicName)` 或者 `QueueSession` 的方法 `createQueue(String queueName)` 来创建一个拥有默认属性的新目的地。依赖于供应商的实现， `DynamicDestinationResolver` 也可能创建一个物理上的目的地，而不是只是解析。

布尔属性 `PubSubDomain` 被用来配置 `JmsTemplate` 使用什么样的 JMS 域。这个属性的默认值是 false，使用点到点的队列。 `JmsTemplate` 使用该属性决定了通过 `DestinationResolver` 的实现来解析动态目的地的行为。

你还可以通过属性 `DefaultDestination` 配置一个带有默认目的地的 `JmsTemplate` 。默认的目的地被使用时，它的发送和接收操作不需要指定一个特定的目的地。

26.2.4 消息监听容器

在 EJB 世界里，JMS 消息最常用的功能之一是用于实现消息驱动 Bean（MDB）。Spring 提供了一个方法来创建消息驱动的 POJO（MDP），并且不会把用户绑定在某个 EJB 容器上。（参见第26.4.2节“异步接收 - 消息驱动的 POJO”，详细介绍了 Spring 的 MDP 支持）。从 Spring Framework 4.1开始，端点方法可以简单使用 @JmsListener 注解，参见第26.6节“注释驱动的侦听器端点“ 更多细节。

用消息监听容器从 JMS 消息队列接收消息，并驱动被注入该消息的消息监听器。监听容器负责消息接收和分发到对应的监听器的所有线程。消息监听容器是 MDP 和消息提供者之间的一个中介，负责处理消息接收的注册、事务管理、资源获取与释放和异常转换等。这使得应用开发人员可以专注于开发和接收消息（可能的响应）相关的（复杂）业务逻辑，把和 JMS 基础框架有关的样板化的部分委托给框架处理。

有两个标准的 JMS 消息监听容器包含在 Spring 中，每一个都有它特殊的功能集。

SimpleMessageListenerContainer

这个消息监听容器是两种标准风格中比较简单的一个，它在启动时创建固定数量的 JMS 会话和消费者，使用标准的 JMS 方法 `MessageConsumer.setMessageListener()` 注册监听，并且让 JMS 提供者做监听回调。它不适于动态运行要求或者参与额外管理事务。兼容上，它与标准的 JMS 规范很近，但它通常情况下不兼容 Java EE 的 JMS 限制条件。

> 虽然 `SimpleMessageListenerContainer` 不允许参与外部管理的事务，但它确实支持原生 JMS 事务：只需将 `sessionTransacted` 标志切换为 true，或者在命名空间中将 `acknowledge` 属性设置为 transacted：监听器抛出的异常将会导致回滚，然后消息被重新传递。或者，考虑使用 `CLIENT_ACKNOWLEDGE` 模式，在异常的情况下提供重新传递，但没有使用事务会话，因此在事务协议中不包括任何其他会话操作（例如发送响应消息）。

DefaultMessageListenerContainer

这个消息监听容器用于大部分的案例中。
与 `SimpleMessageListenerContainer` 相反的是，这个容器适于动态运行要求并且能参与额外管理事务。 在配置 `JtaTransactionManager` 的时候，每一个被接收的消息使用 XA 事务注册，因此可能利用 XA 事务语法处理。该监听容器在 JMS 提供者低要求、高级功能（如外部管理事务的参与）以及与 Java EE 环境的兼容性之间取得了良好的平衡。

容器缓存等级可以定制，注意当缓存不可用的时候，每一次消息接收，一个新的连接和新的会话就会被创建。使用高负载的非持久化订阅可能导致消息丢失，在这种情况下，确保使用合适的缓存等级。

当代理挂掉时，此容器也具备可恢复的能力。默认情况下，一个简单的 `BackOff` 实现会每5秒重试一次。可以为更细粒度的恢复选项指定自定义的 `BackOff` 实现，请参见 `ExponentialBackOff` 示例。

> 与同级的 `SimpleMessageListenerContainer` 一样， `DefaultMessageListenerContainer` 支持原生 JMS 事务，并允许自定义确认模式。如果可行的话，强烈建议您使用外部管理的事务：即，如果你可以忍受在 JVM 挂掉的情况下偶尔会重复发送消息。业务逻辑中的自定义重复消息检测步骤可能涵盖这些情况，例如以业务实体的形式存在的检查或协议表检查。这样的安排将比任何其他方式显着更有效：用 XA 事务（通过使用 `JtaTransactionManager` 配置你的 `DefaultMessageListenerContainer` ）来包裹整个过程，覆盖了 JMS 消息的接收以及消息监听器中业务逻辑的执行（包括数据库操作等）。

26.2.5 事务管理

Spring 提供了一个 `JmsTransactionManager` 用于对 JMS `ConnectionFactory` 做事务管理。这将允许 JMS 应用利用 Spring 的事务管理特性。第13章事务管理中所述的 Spring 的托管事务功能。 `JmsTransactionManager` 在执行本地资源事务管理时将从指定的 `ConnectionFactory` 绑定一个 `ConnectionFactory/Session` 这样的配对到线程中。 `JmsTemplate` 会自动检测这样的事务资源，并对它们进行相应操作。

在 Java EE 环境中， `ConnectionFactory` 会池化连接和会话，这样这些资源将会在整个事务中被有效地重复利用。在一个独立的环境中，使用 Spring 的 `SingleConnectionFactory` 时所有的事务将公用一个JMS 连接，但是每个事务将保留自己独立的会话。或者，请考虑使用具体提供者的池适配器，如 ActiveMQ 的 `PooledConnectionFactory` 类。

`JmsTemplate` 也利用 `JtaTransactionManager` 和支持 XA 的 JMS `ConnectionFactory` 一起来执行分布式事务。请注意，这需要使用 JTA 事务管理器以及正确的 XA 配置的 `ConnectionFactory` ！（检查您的 Java EE 服务/ JMS 提供者的文档。）

在使用 JMS API 从连接中创建会话时，通过托管和非托管事务环境重用代码可能会令人困惑。 这是因为 JMS API 只有一种工厂方法来创建会话，它需要对事务和确认模式赋值。在托管环境中，设置这些值是环境事务性基础架构的责任，因此供应商对 JMS 连接的包装器将忽略这些值。 在非托管环境中使用 `JmsTemplate` 时，可以通过使用属性 `sessionTransacted` 和 `sessionAcknowledgeMode` 来指定这些值。 当与 `JmsTemplate` 一起使用 `PlatformTransactionManager` 时，模板将始终被赋予一个事务性 JMS 会话。

26.3 发送消息

`JmsTemplate` 包含许多方便的方法来发送消息。有些发送方法可以使用 `javax.jms.Destination` 对象指定目的地，也可以使用字符串在 JNDI 中查找目的地。没有目的地参数的发送方法使用默认的目的地。

```
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

    private JmsTemplate jmsTemplate;
    private Queue queue;

    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    public void setQueue(Queue queue) {
        this.queue = queue;
    }

    public void simpleSend() {
        this.jmsTemplate.send(this.queue, new MessageCreator() {
            public Message createMessage(Session session) throws
 JMSException {
                return session.createTextMessage("hello queue wo
rld");
            }
        });
    }
}
```

这个例子使用 `MessageCreator` 回调接口从提供的会话对象中创建一个文本消息，并且通过一个 `ConnectionFactory` 的引用来构造 `JmsTemplate` 。或者，提供了一个无参数的构造方法和 `connectionFactory` ，并可用于以 JavaBean 方式构建实例（使用 BeanFactory 或纯 Java 代码）。或者考虑从 Spring 的基类 `JmsGatewaySupport` 派生，它对 JMS 配置具有内置的 bean 属性。

方法 `send(String destinationName，MessageCreator creator)` 让你利用目的地的字符串名称发送消息。如果这些名称在 JNDI 中注册，则应将模板的 `destinationResolver` 属性设置为 `JndiDestinationResolver` 的一个实例。

如果创建了 `JmsTemplate` 并指定一个默认的目的地，那么 `send(MessageCreator c)` 会向该目的地发送消息。

26.3.1 使用消息转换器

为便于发送领域模型对象， `JmsTemplate` 有多种以一个 Java 对象为参数并做为发送消息的数据内容。 `JmsTemplate` 里可重载的方法 `convertAndSend` 和 `receiveAndConvert` 将转换的过程委托给接口 `MessageConverter` 的一个实例。这个接口定义了一个简单的合约用来在 Java 对象和 JMS 消息间进行转换。缺省的实现 `SimpleMessageConverter` 支持 `String` 和 `TextMessage` ， `byte[]` 和 `BytesMesssage` ,以及 `java.util.Map` 和 `MapMessage` 之间的转换。使用转换器，可以使你和你的应用关注于通过 JMS 接收和发送的业务对象而不用操心它是具体如何表达成 JMS 消息的。

目前的沙箱模型包括一个 `MapMessageConverter` ，它使用反射转换 JavaBean 和 `MapMessage` 。其他流行可选的实现方式包括使用已存在的 XML 编组的包（如 JAXB，Castor 或 XStream）来创建一个表示对象的 `TextMessage` 。

为方便那些不能以通用方式封装在转换类里的消息属性、消息头和消息体的设置，通过 `MessagePostProcessor` 接口，你可以在消息被转换后并且在发送前访问该消息。下例展示了如何在 `java.util.Map` 已经转换成一个消息后更改消息头和属性。

```
public void sendWithConversion() {
    Map map = new HashMap();
    map.put("Name", "Mark");
    map.put("Age", new Integer(47));
    jmsTemplate.convertAndSend("testQueue", map, new MessagePost
Processor() {
        public Message postProcessMessage(Message message) throw
s JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");
            return message;
        }
    });
}
```

这将产生一个如下的消息格式:

```
MapMessage={
    Header={
        ... standard headers ...
        CorrelationID={123-00001}
    }
    Properties={
        AccountID={Integer:1234}
    }
    Fields={
        Name={String:Mark}
        Age={Integer:47}
    }
}
```

26.3.2 SessionCallback和ProducerCallback

虽然 send 操作适用于许多常见的使用场景，但是有时你需要在一个 JMS 会话（ `Session` ） 或者 `MessageProducer` 上执行多个操作。接口 `SessionCallback` 和 `ProducerCallback` 分别提供了 JMS Session 和

Session / MessageProducer 对。在 `JmsTemplate` 上的 `execute()` 方法执行这些回调方法。

26.4 接收消息

26.4.1 同步接收

虽然 JMS 通常与异步处理相关，但它也可以同步地消费消息。可重载的 `receive(..)` 方法提供了这个功能。在同步接收期间，调用线程阻塞，直到接收到消息。这可能是一个危险的操作，因为调用线程可能无限期地被阻塞。`receiveTimeout` 属性指定了接收者等待消息的超时时间。

26.4.2 异步接收 - 消息驱动的 POJOs

> Spring 还可以通过使用 `@JmsListener` 注解来支持监听注解端点，并提供了一种以编程方式注册端点的开放式基础架构。 这是设置异步接收器的最方便的方法，有关详细信息，请参见第26.6.1节“启用监听端点注解”。

类似于 EJB 世界里流行的消息驱动 bean(MDB)，消息驱动 POJO(MDP) 作为 JMS 消息的接收器。MDP 的一个约束(请看下面的有关 `javax.jms.MessageListener` 类的讨论)是它必须实现 `javax.jms.MessageListener` 接口。另外当你的 POJO 将以多线程的方式接收消息时必须确保你的代码是线程安全的。

下面是 MDP 的一个简单实现:

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                System.out.println(((TextMessage) message).getTe
xt());
            }
            catch (JMSException ex) {
                throw new RuntimeException(ex);
            }
        }
        else {
            throw new IllegalArgumentException("Message must be
of type TextMessage");
        }
    }

}
```

一旦你实现了 `MessageListener` 接口,下面该创建一个消息监听容器了。

请看下面例子是如何定义和配置一个随 Sping 发行的消息侦听容器的(这个例子用 `DefaultMessageListenerContainer`)。

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the message listener container -->
<bean id="jmsContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener" />
</bean>
```

请参阅各种消息监听容器的 Spring javadocs，以了解每个实现所支持功能的完整描述。

26.4.3 SessionAwareMessageListener 接口

`SessionAwareMessageListener` 接口是一个 Spring 专门用来提供类似于 JMS `MessageListener` 的接口，也提供了从接收 `Message` 来访问 JMS `Session` 的消息处理方法。

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSE
xception;

}
```

如果你希望你的 MDP 可以响应所有接收到的消息（使用 `onMessage(Message, Session)` 方法提供的 `Session` ）那么你可以选择让你的 MDP 实现这个接口（优先于标准的 JMS `MessageListener` 接口）。所有随 Spring 发行的支持 MDP 的消息监听容器都支持 `MessageListener` 或 `SessionAwareMessageListener` 接口的实现。要注意的是实现了 `SessionAwareMessageListener` 接口的类通过接口与 Spring 有了耦合。是否选择使用它完全取决于开发者或架构师。

请注意`SessionAwareMessageListener`接口的`onMessage(..)`方法会抛出`JMSException`异常。和标准 JMS `MessageListener`接口相反，当使用`SessionAwareMessageListener`接口时，客户端代码负责处理所有抛出的异常。

26.4.4 MessageListenerAdapter

`MessageListenerAdapter`类是 Spring 的异步支持消息类中的最后一个组建：简而言之，它允许您将几乎任何类都暴露为MDP（当然有一些限制）。

请考虑以下接口定义。请注意，虽然该接口既不继承`MessageListener`，也不继承`SessionAwareMessageListener`接口，但通过`MessageListenerAdapter`类依然可以当作一个 MDP 使用。还要注意，各种消息处理方法是如何根据可以接收和处理的各种消息的内容进行强类型匹配的。

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);

}
```

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

尤其要注意的是，上述`MessageDelegate`接口的实现（上述`DefaultMessageDelegate`类）完全不依赖于 JMS。它是一个真正的 POJO，我们可以通过如下配置把它设置成 MDP。

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="org.springframework.jms.listen
er.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the message listener container... -->
<bean id="jmsContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener" />
</bean>
```

以下是另一个只能接收 JMS `TextMessage` 消息的 MDP 示例。注意消息处理方法是如何实际调用 `receive` (在 `MessageListenerAdapter` 中默认的消息处理方法的名字是 `handleMessage`)的，但是它是可配置的(从下面可以看到)。注意 `receive(..)` 方法是如何使用强制类型来只接收和处理JMS `TextMessage` 消息的。

```
public interface TextMessageDelegate {

    void receive(TextMessage message);

}
```

```
public class DefaultTextMessageDelegate implements TextMessageDe
legate {
    // implementation elided for clarity...
}
```

辅助的 `MessageListenerAdapter` 类配置文件类似如下：

```
<bean id="messageListener" class="org.springframework.jms.listen
er.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

请注意，如果上述 `messageListener` 接收到不是 `TextMessage` 类型的 JMS 消息，则会抛出 `IllegalStateException` （随之产生的其他异常只被捕获而不处理）。 `MessageListenerAdapter` 还有一个功能就是如果处理方法返回一个非空值，它将自动返回一个响应消息。请看下面的接口及其实现：

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);

}
```

```
public class DefaultResponsiveTextMessageDelegate implements Res
ponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

如果将上述 `DefaultResponsiveTextMessageDelegate` 与 `MessageListenerAdapter` 联合使用，那么从执行 `receive(..)` 方法返回的任何非空值都将（缺省情况下）转换为 `TextMessage` 。这个返回的 `TextMessage` 将被发送到原来的 `Message` 中 JMS Reply-To 属性定义的目的地（如果存在），或者

是 `MessageListenerAdapter` 设置（如果配置了）的缺省目的地；如果没有定义目的地，那么将产生一个 `InvalidDestinationException` 异常（此异常将不会只被捕获而不处理，它将沿着调用堆栈上传）。

26.4.5 事务中的消息处理

在事务中调用消息监听器只需要重新配置监听容器。

本地资源事务可以通过监听容器上定义的 `sessionTransacted` 标志进行简单地激活。 然后，每个消息监听器调用将在激活的 JMS 事务中进行操作，并在监听器执行失败的情况下进行消息回滚。 发送响应消息（通过 `SessionAwareMessageListener` ）将成为同一本地事务的一部分，但任何其他资源操作（如数据库访问）将独立运行。 在监听器的实现中通常需要进行重复消息的检测，覆盖数据库处理已经提交但消息处理提交失败的情况。

```
<bean id="jmsContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="sessionTransacted" value="true"/>
</bean>
```

对于参与外部管理的事务，你将需要配置一个事务管理器并使用支持外部管理事务的监听容器：通常为 `DefaultMessageListenerContainer` 。

要配置 XA 事务参与的消息监听容器，您需要配置一个 `JtaTransactionManager` （默认情况下，它将委托给 Java EE 服务器的事务子系统）。请注意，底层的 JMS `ConnectionFactory` 需要具有 XA 能力并且正确地注册到你的 JTA 事务协调器上！（检查你的 Java EE 服务的 JNDI 资源配置。）这允许消息接收以及例如同一事务下的数据库访问（具有统一提交语义，以 XA 事务日志开销为代价）。

```
<bean id="transactionManager" class="org.springframework.transac
tion.jta.JtaTransactionManager"/>
```

然后，你只需要将它添加到我们之前的容器配置中。其余的交给容器处理。

```
<bean id="jmsContainer" class="org.springframework.jms.listener.
DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="transactionManager" ref="transactionManager"
/>
</bean>
```

26.5 支持 JCA 消息端点

从 Spring2.5 版本开始，Spring 也提供了基于 JCA `MessageListener` 容器的支持。 `JmsMessageEndpointManager` 将根据提供者 `ResourceAdapter` 的类名自动地决定 `ActivationSpec` 类名。因此，通常它只提供如下例所示的 Spring 的通用 `JmsActivationSpecConfig` 。

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessag
eEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpecConfig">
        <bean class="org.springframework.jms.listener.endpoint.J
msActivationSpecConfig">
            <property name="destinationName" value="myQueue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

或者，您可以使用给定的 `ActivationSpec` 对象设置 `JmsMessageEndpointManager` 。 `ActivationSpec` 对象也可能来自 JNDI 查找（使用 `<jee：jndi-lookup>` ）。

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessag
eEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpec">
        <bean class="org.apache.activemq.ra.ActiveMQActivationSp
ec">
            <property name="destination" value="myQueue"/>
            <property name="destinationType" value="javax.jms.Qu
eue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

使用 Spring 的 `ResourceAdapterFactoryBean` ，目标 `ResourceAdapter` 可以在本地配置，如以下示例所示。

```
<bean id="resourceAdapter" class="org.springframework.jca.support.ResourceAdapterFactoryBean">
    <property name="resourceAdapter">
        <bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl" value="tcp://localhost:61616"/>
        </bean>
    </property>
    <property name="workManager">
        <bean class="org.springframework.jca.work.SimpleTaskWorkManager"/>
    </property>
</bean>
```

指定的 `WorkManager` 也可能指向环境特定的线程池 - 通常通过 `SimpleTaskWorkManager` 的 `asyncTaskExecutor` 属性。如果，你恰好考虑使用多个适配器，为你的所有 `ResourceAdapter` 实例定义一个共享线程池。

在某些环境（例如 WebLogic 9或更高版本）中，可以从 JNDI 中获取整个 `ResourceAdapter` 对象（使用 `<jee：jndi-lookup>` ）。然后，基于Spring的消息监听器可以与服务器托管的 `ResourceAdapter` 进行交互，也可以使用服务内置的 `WorkManager` 。

有关更多详细信息，请参阅 `JMSMessageEndpointManager` 、 `JmsActivationSpecConfig` 和 `` `ResourceAdapterFactoryBean `` 的 JavaDoc。

Spring 还提供了一个通用的 JCA 消息端点管理器，它不绑定到 JMS ： `org.springframework.jca.endpoint.GenericMessageEndpointManager` 。它允许使用任何消息监听器类型（例如 CCI `MessageListener` ）和任何提供者特定的 `ActivationSpec` 对象。从所涉及 JCA 提供者的文档可以找到这个连接器的实际能力，并参考“ `GenericMessageEndpointManager` 的 JavaDoc ”来了解 Spring 特有的配置详细信息。

> 基于 JCA 的消息端点管理器与 EJB 2.1的消息驱动 Bean 很相似；它使用了提供者们约定的相同底层资源。 与 EJB 2.1 MDB 一样，任何被 JCA 提供者支持的消息监听器接口都可以在 Spring 上下文中使用。尽管如此，Spring 仍为 JMS 提供了显式的“方便的”支持，很显然是因为 JMS 是 JCA 端点管理约定中最通用的端点 API。

26.6 注解驱动的监听端点

异步接收消息的最简单的方法是使用注解监听端点的基础架构。简而言之，它允许你暴露托管一个 bean 的方法作为一个 JMS 的监听端点。

```
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(String data) { ... }
}
```

上述示例的想法是，每当 `javax.jms.Destination` “myDestination” 上有消息可用时，就调用用相应地 `processOrder` 方法（在这种情况下，JMS消息的内容类似于 `MessageListenerAdapter` 提供的内容）。

注解端点的基础架构使用 `JmsListenerContainerFactory` 为每个注解方法创建一个消息监听容器。这样的容器没有针对应用上下文进行注册，但是可以使用 `JmsListenerEndpointRegistry` bean 进行简单的管理。

> `@JmsListener` 是 Java 8上的可重复注解，因此可以通过向其添加额外的 `@JmsListener` 声明将多个JMS目的地关联到同一个方法。 在 Java 6和7上，你可以使用 `@JmsListeners` 注解。

26.6.1 启用监听端点的注解

要启用对 `@JmsListener` 注解的支持，请将 `@EnableJms` 添加到你的一个 `@Configuration` 类上。

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContain
erFactory() {
        DefaultJmsListenerContainerFactory factory =
                new DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        factory.setDestinationResolver(destinationResolver());
        factory.setConcurrency("3-10");
        return factory;
    }
}
```

默认情况下，基础架构将查找名为 `jmsListenerContainerFactory` 的 bean 作为用于创建消息监听容器的工厂源。 在这种情况下，忽略 JMS 基础架构的设置， `processOrder` 方法可以在一个线程池中被调用，线程池核心数为3，最大线程数为10。

对于使用的每个注解都可以自定义监听容器工厂，或通过实现 `JmsListenerConfigurer` 接口来显示的配置默认值。仅在存在没有指定容器工厂的端点时，默认值才是必须的。有关详细信息和示例，请参阅 javadoc。

如果您喜欢XML配置，请使用 `<jms：annotation-driven>` 元素。

```
<jms:annotation-driven/>

   <bean id="jmsListenerContainerFactory"
           class="org.springframework.jms.config.DefaultJmsListe
nerContainerFactory">
       <property name="connectionFactory" ref="connectionFactory
"/>
       <property name="destinationResolver" ref="destinationReso
lver"/>
       <property name="concurrency" value="3-10"/>
   </bean>
```

26.6.2 编程式端点注册

`JmsListenerEndpoint` 提供了 JMS 端点的模型，并负责为该模型配置容器。除了使用注解之外，基础架构也允许你用编程的方式来配置端点。

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistr
ar registrar) {
        SimpleJmsListenerEndpoint endpoint = new SimpleJmsListen
erEndpoint();
        endpoint.setId("myJmsEndpoint");
        endpoint.setDestination("anotherQueue");
        endpoint.setMessageListener(message -> {
            // processing
        });
        registrar.registerEndpoint(endpoint);
    }
}
```

本例中我们使用 `SimpleJmsListenerEndpoint` 来提供 `MessageListener` ，你也可以建立自己的端点变体并自定义调用机制。

应该注意的是，你完全可以不使用 `@JmsListener` ，而仅通过 `JmsListenerConfigurer` 来注册所有端点。

26.6.3 注解式端点方式签名

到目前为止，我们已经在我们的端点注入了一个简单的 `String` ，但实际上它可以有一个非常灵活的方法签名。现在让我们重写它并注入一个带有自定义头部的 `Order` ：

```
@Component
public class MyService {

        @JmsListener(destination = "myDestination")
        public void processOrder(Order order, @Header("order_type
") String orderType) {
                ...
        }
}
```

可以向 JMS 监听端点中注入的主要元素包括：

- 原始的 `javax.jms.Message` 或任意子类（当然，它与传入的消息类型相匹配）。
- 可选的 `javax.jms.Session` 来操作 JMS 原生 API，来发送自定义回复。
- 代表着接收消息的 `org.springframework.messaging.Message` 。注意此消息同时包含自定义和 `JmsHeaders` 定义的标准头。
- `@Header` 注解的方法参数被用来提取一个特定的头部值，包括标准 JMS 头。
- `@Headers` 注解参数必须指定给一个 `java.util.Map` ，用来获取所有头。
- 不被支持的（如 `Message` 、 `Session` 等）、且无注解的元素被视为有效载荷。可以明确的给它们添加 `@Payload` 注解。也可以通过添加 `@Valid` 注解来开启校验。

注入 Spring 的 `Message` 抽象可以获取特定消息的所有信息，而无需依赖特定传输 API。

```
@JmsListener(destination = "myDestination")
public void processOrder(Message<Order> order) { ... }
```

可以自己扩展 `DefaultMessageHandlerMethodFactory` 来处理额外的方法参数。同时你也可以自定义转换和校验规则。

例如，如果我们需要在处理之前确保我们的 `Order` 有效，我们可以使用 `@Valid` 对有效负载进行注解，并配置必要的验证器，如下所示：

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

        @Override
        public void configureJmsListeners(JmsListenerEndpointRegi
strar registrar) {
            registrar.setMessageHandlerMethodFactory(myJmsHandler
MethodFactory());
        }

        @Bean
        public DefaultMessageHandlerMethodFactory myHandlerMethod
Factory() {
            DefaultMessageHandlerMethodFactory factory = new Defa
ultMessageHandlerMethodFactory();
            factory.setValidator(myValidator());
            return factory;
        }
}
```

26.6.4 响应管理

`MessageListenerAdapter` 允许你的方法有非空返回值。此时返回值将被封装在一个 `javax.jms.Message` 中，发往原始消息的 `JMSReplyTo` 头中定义的目的地或者监听自己默认的目的地。监听默认的目的地可以通过 `@SendTo` 注解来设置。

假定现在我们的 `processOrder` 方法将返回一个 `OrderStatus` ，下面将展示如何自动地发送响应：

```
@JmsListener(destination = "myDestination")
@SendTo("status")
public OrderStatus processOrder(Order order) {
        // order processing
        return status;
}
```

> 如果你有多个 `@JmsListener` 注解方法，您还可以将 `@SendTo` 注解放在 class 上以共享默认响应目的地。

如果您需要以独立传输的方式设置额外的头信息，则可以返回一个 `Message` ，如下所示：

```
@JmsListener(destination = "myDestination")
@SendTo("status")
public Message<OrderStatus> processOrder(Order order) {
	// order processing
	return MessageBuilder
			.withPayload(status)
			.setHeader("code", 1234)
			.build();
}
```

如果响应的目的地是运行时实时计算的，可以将响应结果封装在一个 `JmsResponse` 中，直接指定一个目的地。前面的例子可以重写如下：

```
@JmsListener(destination = "myDestination")
public JmsResponse<Message<OrderStatus>> processOrder(Order orde
r) {
	// order processing
	Message<OrderStatus> response = MessageBuilder
			.withPayload(status)
			.setHeader("code", 1234)
			.build();
    return JmsResponse.forQueue(response, "status");
}
```

26.7 JMS 命名空间的支持

Spring 引入了 XML 命名空间以简化 JMS 的配置。使用 JMS 命名空间元素时，需要引用如下的 JMS Schema：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jms="http://www.springframework.org/schema/jms"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans http://w
ww.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/jms http://www
.springframework.org/schema/jms/spring-jms.xsd">

    <!-- bean definitions here -->

</beans>
```

命名空间由三个顶级元素组成：，和。可以使用注解驱动的监听端点。 和定义共享监听容器的配置，并且包含了子元素。下面是一个基本配置的示例，包含两个监听器。

```
<jms:listener-container>

    <jms:listener destination="queue.orders" ref="orderService"
method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirm
ationLogger" method="log"/>

</jms:listener-container>
```

上面的例子等同于在第26.4.4节“MessageListenerAdapter”的示例，定义两个不同的监听器容器和两个不同的 `MessageListenerAdapter` 。除了上面显示的属性之外， `listener` 元素也包含几个可选属性。下面的表格列出了所有的属性：

表26.1 JMS \的元素属性

属性	描述
id	托管监听容器的 bean 名称。如果未指定，将自动生成一个 bean 名称。
destination（必选）	监听器的目的地名称，由 `DestinationResolver` 的策略决定。
ref（必选）	处理对象的 bean 名称。
method	处理器中被调用的方法名。如果 ref 指向 `MessageListener` 或者 Spring `SessionAwareMessageListener` ，则这个属性可以被忽略。
response-destination	默认的响应目的地是发送响应消息抵达的目的地。 这用于请求消息没有包含 `JMSReplyTo` 域的情况。响应目的地类型被监听容器的 `destination-type` 属性决定。记住：这仅仅适用于有返回值的监听器方法，因为每个结果对象都会被转化成响应消息。
subscription	持久订阅的名称（如果需要的话）。
selector	监听器的一个可选的消息选择器。
concurrency	监听器启动的会话/消费者的并发数量。可以是表示最大数量（例如“5”）的简单数字，也可以是表示下限以及上限（例如“3-5”）的范围。 请注意，指定的最小值只是一个提示，在运行时可能会被忽略。默认值是容器提供的值。

\元素也接受几个可选属性。这允许自定义各种策略（例如， `taskExecutor` 和 `destinationResolver` ）以及基本的 JMS 设置和资源引用。使用这些属性，可以定义很广泛的定制监听容器，同时仍享有命名空间的便利。

作为一个通过 `factory-id` 属性指定要暴露的 bean 的 id 的 `JmsListenerContainerFactory` ，自动暴露了这些设置。

```
jms:listener-container connection-factory="myConnectionFactory"
        task-executor="myTaskExecutor"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="orderService"
method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirm
ationLogger" method="log"/>

</jms:listener-container>
```

下面的表格描述了所有可用的属性。参考 `AbstractMessageListenerContainer` 类和具体子类的 Javadoc 来了解每个属性的细节。这部分的 Javadoc 也提高那个了事务选择和消息传输场景的讨论。

表26.2 JMS \的元素属性

属性	描述
container-type	监听容器的类型。 可用的选项有：default，simple，default102或 为"default"）。
container-class	自定义监听容器的实现类作为完全限定类名。默认是 Spring 的标准 `DefaultMessageListenerContainer` 或 `SimpleMessageLis` 取决于 `container-type` 属性。
factory-id	通过对 `JmsListenerContainerFactory` 指定 id，暴露该元素的 与其他端点一起使用。
connection-factory	对 JMS `ConnectionFactory` bean 的引用（默认 bean 名称为 `connectionFactory` ）。
task-executor	对JMS监听器调用者的 Spring `TaskExecutor` 的引用。
destination-resolver	对解决 JMS 目标的 `DestinationResolver` 策略的引用。
message-converter	对 JMS 消息转换为监听器方法参数的 `MessageConverter` 策略 个 `SimpleMessageConverter` 。
error-handler	对处理任何未捕获异常的 `ErrorHandler` 策略的引用，异常可能

error-handler	在 `MessageListener` 的执行期间。
destination-type	监听器的 JMS 目标类型：queue，topic，durableTopic，shared sharedDurableTopic。 这样可以间接启用容器的 `pubSubDomain` ， `subscriptionDurable` 和 `subscriptior` 是队列（即禁用这3个属性）。
response-destination-type	响应的JMS目标类型：queue、topic。 默认值为 `destination-t`
client-id	监听容器的 JMS 客户 ID。 使用持久订阅时需要指定。
cache	JMS资源的缓存级别：none, connection, session, consumer 或者（auto），缓存级别有效的是“consumer”。除非已经指定了外部 情况下，有效的默认值为 none（假设 Java EE 风格的事务管理，的 `ConnectionFactory` 是 XA-aware 池）。
acknowledge	原生JMS确认模式：auto，client，dups-ok 或 transacted。trans 交易的会话。 或者，指定下面描述的 `transaction-manager` 属
transaction-manager	对外部 `PlatformTransactionManager` 的引用（通常是基于XA Spring 的 `JtaTransactionManager` ）。如果未指定，将使用本 阅 `acknowledge` 属性）。
concurrency	每个监听器启动的会话/消费者的并发数量。 可以是表示最大数量 数字，也可以是表示下限以及上限（例如“3-5”）的范围。 请注意 一个提示，在运行时可能会被忽略。 默认值为1；在 topic 监听器 很重要的情况下，将并发限制为1；一般的queue可以考虑提高并
prefetch	要加载到单个会话的消息的最大数量。请注意，提高此数量可能 饥饿！
receive-timeout	用于接收调用的超时时间（以毫秒为单位）。 默认值为1000 ms 超时限制。
back-off	指定用于计算恢复尝试间隔的 `BackOff` 实例。 如果 `BackOffEx` 回 `BackOffExecution#STOP` ，监听容器将不再进一步尝试恢复 时，将忽略 `recovery-interval` 值。 默认值为 `FixedBackOff` ms，即5秒。
recovery-interval	指定恢复尝试之间的间隔（以毫秒为单位）。是以指定间隔创建 捷方式。 有关更多恢复选项，请考虑指定 `BackOff` 实例。 默认 秒。
phase	此容器应在其中开始和停止的生命周期阶段。 值越小，容器就越 止。 默认值为 `Integer` 。 `MAX_VALUE` ，意味着容器将尽可能 止。

使用 jms Schema 支持来配置基于 JCA 的监听器容器很相似。

```
<jms:jca-listener-container resource-adapter="myResourceAdapter"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="myMessageListe
ner"/>

</jms:jca-listener-container>
```

JCA 可用的配置选项描述如下表：

表26.3 JMS \的元素属性

属性	描述
factory-id	
resource-adapter	对 JCA `ResourceAdapter` bean 的引用（默认 bean 名称是 `resourceAdapter` ）。
activation-spec-factory	对 `JmsActivationSpecFactory` 的引用。默认情况是自动检测及其 `ActivationSpec` 类（请参阅 `DefaultJmsActivationSpe`
destination-resolver	对解决JMS目标的 `DestinationResolver` 策略的引用。
message-converter	对 JMS 消息转换为监听器方法参数的 `MessageConverter` 策略是一个 `SimpleMessageConverter` 。
error-handler	对处理任何未捕获异常的 `ErrorHandler` 策略的引用，异常可能在 `MessageListener` 的执行期间。
destination-type	监听器的JMS目标类型：queue，topic，durableTopic，sharedTo sharedDurableTopic。这样可以间接启用容器的pubSubDomain subscriptionDurable和subscriptionShared属性。默认是队列（即性）。
response-destination-type	响应的JMS目标类型："queue"，"topic"。默认值为"destination-t值。
client-id	监听容器的 JMS 客户 ID。使用持久订阅时需要指定。
acknowledge	原生JMS确认模式：auto，client，dups-ok 或 transacted。trans了本地交易的会话。或者，指定下面描述的 `transaction-mana` 默认为 auto。
transaction-manager	对 Spring `JtaTransactionManager` 或 `javax.transaction.Transacti` 引用，为每个传入消息启动 XA 事务。如果未指定，将使用本地阅 `acknowledge` 属性）。
concurrency	每个监听器启动的会话/消费者的并发数量。可以是表示最大数量的简单数字，也可以是表示下限以及上限（例如"3-5"）的范围。的最小值只是一个提示，在使用 JCA 监听容器的运行时可能会被值为1;
prefetch	要加载到单个会话的消息的最大数量。请注意，提高此数量可能费者的饥饿！

27 JMX

27.1 引言

Spring对JMX的支持提供了你可以简单、透明的将Spring应用程序集成到JMX的基础架构中。

```
JMX?

本章不是介绍JMX的...它不会试图去解释为什么要使用JMX（或JMX实际代表什么含义）的动机。如果你是JMX的新手，请参考本章末尾的[第27.8节，更多资源](jmx.html#jmx-resources)。
```

具体来说，Spring JMX支持提供了四个核心功能：

- 任何Spring bean都会自动注册为JMX MBean
- bean管理接口的灵活控制机制
- 可以通过JSR-160连接器将声明的MBeans暴露给远程
- 远程和本地MBean资源的简单代理

这个功能的设计是应用程序组件在和Spring或JMX接口和类无需耦合的方式工作。事实上，在大多数情况下应用程序为了使用Spring JMX的特性，也不会去关心Spring或者JMX。

27.2 将Bean暴露给JMX

MBeanExporter是Spring JMX 框架中的核心类。它负责把Spring bean注册到JMX MBeanServer。例如，下面的例子：

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

        private String name;
        private int age;
        private boolean isSuperman;

        public int getAge() {
                return age;
        }

        public void setAge(int age) {
                this.age = age;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public int add(int x, int y) {
                return x + y;
        }

        public void dontExposeMe() {
                throw new RuntimeException();
        }
}
```

为了将此bean的属性和方法作为一个属性和MBean的操作暴露出来，你只需简单的在配置文件中配置MBeanExporter类的实例并把此bean传递进去，如下：

```
<beans>
    <!-- this bean must not be lazily initialized if the exporti
ng is to happen -->
    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter" lazy-init="false">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-r
ef="testBean"/>
            </map>
        </property>
    </bean>
    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

上面配置段中相关的bean定义就是导出的bean。beans的属性准确的告诉MBeanExporter，哪个bean必须暴露给JMX MBeanServer。默认配置，在beans Map中的每个key都是相应的value引用的bean的对象名称。可以像27.4中“控制bean的对象名称”描述的那样来修改此行为。 在这个配置下testBean被暴露为一个名字为bean:name=testBean1的MBean。默认，bean的所有公共的属性都会被暴露为属性并且所有的公共方法（那些从Object类继承的）都被会暴露为操作。

> MBeanExporter是一个有生命周期bean（参考“启动和关闭回调”）并且MBeans会在应用程序默认的生命周期中尽可能迟的被暴露。可以通过在暴露的阶段配置或者通过设置自动启动标志位来禁止自动注册。

27.2.1 创建MBeanServer

假设上述的配置是在一个正在运行的应用环境中，并且他有一个(只有一个)MBeanServer已经在运行了。在这种情形下，Spring将会尝试定位正在运行的MBeanServer并把你的bean注册到它上面去。当你的应用是运行在一个诸如

Tomcat、IBM WebSphere等有自己MBeanServer的容器内时，这个就非常有用。

然而这种方法在独立的环境或运行在一个没有提供MBeanServer的容器中是没用的。为了解决这个问题，你可以通过添加一个org.springframework.jmx.support.MBeanServerFactoryBean的类实例到你的配置中来创建一个MBeanServer实例。你还可以通过将MBeanServerFactoryBeanMBean返回的MBeanServer值赋给MBeanExporter的server属性使用指定的MBeanServer，例如：

```
<beans>

    <bean id="mbeanServer" class="org.springframework.jmx.suppor
t.MBeanServerFactoryBean"/>

    <!--
    this bean needs to be eagerly pre-instantiated in order for
the exporting to occur;
    this means that it must not be marked as lazily initialized
    -->
    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
            <property name="beans">
                    <map>
                            <entry key="bean:name=testBean1" val
ue-ref="testBean"/>
                    </map>
            </property>
            <property name="server" ref="mbeanServer"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
            <property name="name" value="TEST"/>
            <property name="age" value="100"/>
    </bean>

</beans>
```

这是通过MBeanServerFactoryBean创建的一个MBeanServer实例，并通过server属性把它传递给MBeanExporter。当你自己提供MBeanServer实例时，MBeanExporter不在尝试寻找正在运行的MBeanServer，将直接使用提供的MBeanServer实例。为了让它正常运行，你必须（当然）在你的类路径下有一个JMX的实现。

27.2.2重用存在的MBeanServer

如果没有指明的server，那么MBeanExporter会尝试去自动探测一个正在运行的MBeanServer。这在大多数情况下只有一个MBeanServer实例的情况下正常运行，但是当有多个实例存在时，exporter可能会选择一个错误的server。在这种情况下，应该要指明你要用的MBeanServer agentId：

```
<beans>
    <bean id="mbeanServer" class="org.springframework.jmx.suppor
t.MBeanServerFactoryBean">
        <!-- indicate to first look for a server -->
        <property name="locateExistingServerIfPossible" value="t
rue"/>
        <!-- search for the MBeanServer instance with the given
agentId -->
        <property name="agentId" value="MBeanServer_instance_age
ntId>"/>
    </bean>
    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="server" ref="mbeanServer"/>
        ...
    </bean>
</beans>
```

对于平台化或一些情形来说，应该使用factory-method来查找动态变化（未知）的MBeanServer的agentId：

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="server">
                <!-- Custom MBeanServerLocator -->
                <bean class="platform.package.MBeanServerLocator
" factory-method="locateMBeanServer"/>
        </property>
    </bean>

    <!-- other beans here -->

</beans>
```

27.2.3MBeans懒加载

如果你用MBeanExporter配置了一个bean，那么它也配置了延迟初始化，MBeanExporter在不破坏他们关联性的情况下会避免bean的实例化。相反，它会注册一个MBeanServer的代理，直到第一次通过代理向容器获取bean的时候才会初始化。

27.2.4MBeans自动注册

任何通过MBeanExporter暴露的bean都是一个有效的MBean，它和MBeanServer注一样，不需要Spring的介入。可以将MBeanExporter的autodetect属性设置为true，MBeans就可以自动被发现：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanE
xporter">
        <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.ex
port.TestDynamicMBean"/>
```

所有的被称为spring:mbean=true都是有效的JMX MBean，它会被Spring自动注册。默认，bean的名字会被用作ObjectName，JMX注册时自动被发现。这个动作可以被重写，详情参考27.4，“控制bean的ObjectNames”。

27.2.5控制注册的动作

考虑Spring MBeanExporter试图使用'bean:name=testBean1'作为ObjectName将MBean注册到MBeanServer的场景。如果一个MBean的实例已经注册了相同的名字，默认情况下这此注册行为将失败(抛出InstanceAlreadyExistsException异常)。可以严格控制MBean在注册到MBeanServer上何时发生了什么的行为。Spring的JMX支持三种不同的注册行为来控制在注册过程中发现MBean有相同的ObjectName；这些注册方法总结如下：

- 表27.1 注册方法

注册方法	解释
REGISTRATION_FAIL_ON_EXISTING	这是默认的注册方法。如果一个MBean实例已经被注册了相同的ObjectName，这个MBean不能注册，并且抛出InstanceAlreadyExistsException异常。已经存在的MBean不受影响。
REGISTRATION_IGNORE_EXISTING	如果一个MBean实例已经被注册了相同的ObjectName，这个MBean不能注册。已经存在的MBean不受影响，也没有异常抛出。这个设置在多个应用之间共享MBeanServer，共享MBean时非常有用。
REGISTRATION_REPLACE_EXISTING	如果一个MBean实例已经被注册了相同的ObjectName，之前存在的MBean将被没有注册的新MBean原地替换(新的MBean替换前一个)。

上面的值是定义在MBeanRegistrationSupport类中的常量（它是MBeanExporter的父类）。|如果你想改变默认的注册行为，你只需要简单的在MBeanExporter的registrationBehaviorName属性上设置上面定义的值即可。

下面的示例说明了怎样用REGISTRATION_REPLACE_EXISTING改变默认的注册行为：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
                <map>
                        <entry key="bean:name=testBean1" value-r
ef="testBean"/>
                </map>
        </property>
        <property name="registrationBehaviorName" value="REGISTR
ATION_REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

27.3 bean的控制管理接口

在前面的例子中，每个已经被暴露为JMX属性和操作的bean的所有public属性和方法，你都可以通过bean的管理接口来控制。你可以精确的控制你所暴露的bean上的哪个属性和方法作为JMX的属性和操作，Spring JMX提供了全面的、可扩展的机制来控制bean的管理接口。

27.3.1 MBeanInfoAssembler接口

底层实现上，MBeanExporter委托了一个org.springframework.jmx.export.assembler.MBeanInfoAssembler接口的实现来负责在每个已经被暴露的bean上定义管理接口。默认实现是org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler，对所有public的属性和方法（如你在前面例子中看到的）都会简单的定义一个管理接口。Spring对MBeanInfoAssembler接口提供了两种额外的实现，它允许使用源码级别的元数据或任意接口来控制管理接口的生成。

27.3.2 源码级别的元数据（Java注解）

使用MetadataMBeanInfoAssembler你可以对bean在源码级别上定义管理接口。org.springframework.jmx.export.metadata.JmxAttributeSource接口封装了元数据的读取。Spring JMX提供了使用Java注解提供了一个名为org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource的默认实现。为了MetadataMBeanInfoAssembler必须配置一个实现了JmxAttributeSource接口的实例才能正常工作（没有默认）。

为了将一个bean标记为JMX的bean，你应该使用ManagedResource类级别的注解。每个你想要暴露为操作的方法都必须用ManagedOperation注解标记，每个想暴露的属性的都必须用ManagedAttribute注解标记。当标记属性的时候你可以忽略getter或setter然后自己创建一个只读或只写的属性。

> 注意 一个被ManagedResource注解的bean，它暴露的操作或属性必须是public的。

下面的例子展示了用注解实现的JmxTestBean类：

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource
;
import org.springframework.jmx.export.annotation.ManagedOperatio
n;
import org.springframework.jmx.export.annotation.ManagedAttribut
e;

@ManagedResource(
                objectName="bean:name=testBean4",
                description="My Managed Bean",
                log=true,
                logFile="jmx.log",
                currencyTimeLimit=15,
                persistPolicy="OnUpdate",
                persistPeriod=200,
                persistLocation="foo",
                persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

        private String name;
        private int age;

        @ManagedAttribute(description="The Age Attribute", curre
ncyTimeLimit=15)
        public int getAge() {
                return age;
        }

        public void setAge(int age) {
                this.age = age;
        }

        @ManagedAttribute(description="The Name Attribute",
                        currencyTimeLimit=20,
                        defaultValue="bar",
                        persistPolicy="OnUpdate")
        public void setName(String name) {
```

```
                this.name = name;
        }

        @ManagedAttribute(defaultValue="foo", persistPeriod=300)
        public String getName() {
                return name;
        }

        @ManagedOperation(description="Add two numbers")
        @ManagedOperationParameters({
                @ManagedOperationParameter(name = "x", descripti
on = "The first number"),
                @ManagedOperationParameter(name = "y", descripti
on = "The second number")})
        public int add(int x, int y) {
                return x + y;
        }

        public void dontExposeMe() {
                throw new RuntimeException();
        }

}
```

可以看到JmxTestBean类被配置了一系列属性的ManagedResource注解标记。这些属性可以通过MBeanExporter产生配置了各种切面的MBean，详情请参考后面的27.3.3节，”源码级别的元数据类型“。

你也注意到了age和name属性都用了ManagedAttribute注解，但是在age属性上，只有getter上使用了。这会使得所有的属性在management接口中都作为属性，只有age属性是只读的。

最终，你会注意到，add(int, int)方法被标记为ManagedOperation属性，而dontExposeMe()这个方法却不是。当使用MetadataMBeanInfoAssembler时，管理接口只包含一个add（int，int）操作。

下面的配置展示了在使用MetadataMBeanInfoAssembler时如何配置MBeanExporter：

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
            <property name="assembler" ref="assembler"/>
            <property name="namingStrategy" ref="namingStrategy"
/>
            <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
                    class="org.springframework.jmx.export.annota
tion.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metad
ata -->
    <bean id="assembler"
                    class="org.springframework.jmx.export.assemb
ler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource
"/>
    </bean>

    <!-- will pick up the ObjectName from the annotation -->
    <bean id="namingStrategy"
                    class="org.springframework.jmx.export.naming
.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource
"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.Annotatio
nTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

这里你将看到MetadataMBeanInfoAssembler被配置为AnnotationJmxAttributeSource的一个实例，并且通过assembler属性传递给MBeanExporter。这对于利用Spring暴露的MBean元数据驱动的管理接口来说是必须的。

27.3.3 源码级别的元数据类型

使用Spring JMX有以下几种源码级别的元数据类型：

- 表 27.2. 源码级别的元数据类型

目标	注解	注解类型
把所有类的实例作为JMX管理的资源	@ManagedResource	类
将一个方法作为JMX的操作	@ManagedOperation	方法
将getter或setter标识为部分的JMX属性	@ManagedAttribute	方法（只限getter或setter）
对一个操作参数定义描述符	@ManagedOperationParameter和@ManagedOperationParameters	方法

使用这些源码级别的元数据类型有以下的配置参数：

参数	描述	应用
ObjectName	使用元数据命名策略决定管理资源的ObjectName	ManagedResource
description	设置友好的资源、属性和操作的描述	ManagedResource, ManagedAttribute, ManagedOperation, ManagedOperationParameter
currencyTimeLimit	设置currencyTimeLimit描述字段值	ManagedResource, ManagedAttribute
defaultValue	设置defaultValue描述字段值	ManagedAttribute
log	设置log描述字段值	ManagedResource
logFile	设置logFile描述字段值	ManagedResource
persistPolicy	置persistPolicy描述字段值	ManagedResource
persistPolicy	设置persistPolicy描述字段值	ManagedResource
persistPeriod	设置persistPeriod描述字段值	ManagedResource
persistLocation	设置persistLocation描述字段值	ManagedResource
persistName	设置persistName描述字段值	ManagedResource
name	设置操作参数展示的名字	ManagedOperationParameter
index	设置操作参数的索引	ManagedOperationParameter

27.3.4 AutodetectCapableMBeanInfoAssembler接口

为了进一步简化配置，Spring引入了继承了MBeanInfoAssembler接口来支持MBean资源的自动发现的AutodetectCapableMBeanInfoAssembler接口。如果你用AutodetectCapableMBeanInfoAssembler的实例配置了MBeanExporter，那么你可以对暴露给JMX的bean进行“投票”。

AutodetectCapableMBeanInfo的唯一实现是MetadataMBeanInfoAssembler，它将对被标记为ManagedResource的属性进行投票。默认使用bean名字作为ObjectName的配置为：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <!-- notice how no 'beans' are explicitly configured her
e -->
        <property name="autodetect" value="true"/>
        <property name="assembler" ref="assembler"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="assembler" class="org.springframework.jmx.export.a
ssembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource">
                <bean class="org.springframework.jmx.export.anno
tation.AnnotationJmxAttributeSource"/>
        </property>
    </bean>

</beans>
```

注意，在这个配置中没有bean被传递给MBeanExporter；然而，JmxTestBean在被标记为ManagedResource属性，而且MetadataMBeanInfoAssembler会发现JmxTestBean并且投票把它包含进来时就被注册了。这种方法唯一问题是

JmxTestBean的名字现在有业务含义。你可以通过更改ObjectName创建的默认行为来解决此问题，如27.4，“控制bean的ObjectName”。

27.3.5使用Java接口定义管理接口

除了MetadataMBeanInfoAssembler，Spring也提供了MetadataMBeanInfoAssembler，Spring也包含了InterfaceBasedMBeanInfoAssembler，它允许约束方法和用一些基于集合接口上定义的方法所暴露出来的属性。 虽然标准的暴露机制是使用接口和一个简单命名方案，InterfaceBasedMBeanInfoAssembler通过去除命名约定来继承这个功能，允许你使用多个的接口，并且不需要bean实现MBean接口。 考虑你先前用来对JmxTestBean定义管理接口的接口：

```
public interface IJmxTestBean {

        public int add(int x, int y);

        public long myOperation();

        public int getAge();

        public void setAge(int age);

        public void setName(String name);

        public String getName();

}
```

这个接口定义的方法和属性将会在JMX的MBean上暴露为操作和属性。下面的代码展示了如何用管理接口定义的接口来配置Spring JMX：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                    <entry key="bean:name=testBean5" value-ref="
testBean"/>
            </map>
        </property>
        <property name="assembler">
            <bean class="org.springframework.jmx.export.assemble
r.InterfaceBasedMBeanInfoAssembler">
                <property name="managedInterfaces">
                    <value>org.springframework.jmx.IJmxTestBean<
/value>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

这里，你可以看到当对任何bean构建管理接口时会使用IJmxTestBean配置InterfaceBasedMBeanInfoAssembler。通过InterfaceBasedMBeanInfoAssembler处理的bean是不需要实现用来产生JMX管理接口的接口，明白这点很重要。 上面的例子中，所有bean的管理接口都是使用IJmxTestBean接口来构建。在许多情况下，不同的bean会使用不同的接口，而不是上面的单一行为。此例中，你可以通过interfaceMappings属性，来传递InterfaceBasedMBeanInfoAssembler的实例，这样key就是bean名字，value就是在这个bean上的用逗号分隔的方法列表。 如果即

没有通过managedInterfaces或interfaceMappings属性指明一个管理接口，那么InterfaceBasedMBeanInfoAssembler会通过反射在使用所有实现了该bean的接口来创建一个管理接口。

27.3.6 使用MethodNameBasedMBeanInfoAssembler

MethodNameBasedMBeanInfoAssembler允许你指定一个方法名列表给要暴露给JMX的属性和操作。配置如下代码：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanE
xporter">
    <property name="beans">
        <map>
            <entry key="bean:name=testBean5" value-ref="testBean
"/>
        </map>
    </property>
    <property name="assembler">
        <bean class="org.springframework.jmx.export.assembler.Me
thodNameBasedMBeanInfoAssembler">
            <property name="managedMethods">
                <value>add,myOperation,getName,setName,getAge</v
alue>
            </property>
        </bean>
    </property>
</bean>
```

上面实例你可以看到，增加了暴露给JMX操作的方法myOperation和getName、setName(String) 和 getAge()等JMX的属性。在上面的代码中，暴露给JMX的方法都和bean有映射的关系。为了控制一个bean一个bean的暴露，使用MethodNameMBeanInfoAssembler的methodMappings属性来映射bean的名字到方法名字列表上。

27.4控制bean的ObjectNames

在底层，MBeanExporter委托了一个ObjectNamingStrategy的实现来获取每个bean在注册时的ObjectName。默认实现为KeyNamingStrategy，ObjectName作为为bean Map的key。而且，KeyNamingStrategy可以将bean Map的key映射到一个属性文件来解析ObjectName。除此KeyNamingStrategy之外，Spring还提供了两个额外的ObjectNamingStrategy实现：一个基于bean的JVM标识来构建ObjectName的IdentityNamingStrategy和使用代码级元数据来获取ObjectName的MetadataNamingStrategy。

27.4.4 从属性中读取ObjectName

你可以配置自己的KeyNamingStrategy实例，并从一个配置的属性实例中读取ObjectName，而不是bean 的key。KeyNamingStrategy将尝试使用bean对应的key来找出它在Properties中的入口。如果没有找到入口或Properties实例为空，那么bean将使用它自己的key。 下面的代码展示了KeyNamingStrategy的一个简单配置：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy" class="org.springframework.jmx.exp
ort.naming.KeyNamingStrategy">
        <property name="mappings">
            <props>
                <prop key="testBean">bean:name=testBean1</prop>
            </props>
        </property>
        <property name="mappingLocations">
            <value>names1.properties,names2.properties</value>
        </property>
    </bean>

</beans>
```

这里KeyNamingStrategy实例是用一个Properties实例来配置的，Properties是由mapping属性定义的Properties实例和位于mapping属性定义的目录下的属性文件合并而成。在这个配置中，testBean的ObjectName由bean:name=testBean1定义，因此它是bean的key对应的属性实例的入口。 如果Properties实例中找不到入口，那么bean的key将是它的ObjectName。

27.4.2 使用MetadataNamingStrategy

MetadataNamingStrategy使用每个bean上ManagedResource属性的objectName属性来创建ObjectName。下面代码展示了MetadataNamingStrategy的配置：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="attributeSource"/>
    </bean>

    <bean id="attributeSource"
                        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</beans>
```

如果ManagedResource属性没有提供objectName，那么将会用下面的格式创建ObjectName：[fully-qualified-package-name]:type=[short-classname],name=[bean-name]。例如，下面的bean产生的ObjectName为：com.foo:type=MyClass,name=myBean。

```
<bean id="myBean" class="com.foo.MyClass"/>
```

27.4.3基于MBean导出的注解配置

如果你更喜欢使用基于注解的方式来定义management接口，则可以很方便的使用MBeanExporter的一个子类：AnnotationMBeanExporter。当定义一个子类的实例时，将不需要再配置namingStrategy，assembler和attributeSource，因为它会使用标注的基于Java 元数据的注解（自动检测一直开启）。实际上，不是定义一个MBeanExporter，@EnableMBeanExport和@Configuration注解支持更简单的语法。

```
@Configuration@EnableMBeanExport
public class AppConfig {

}
```

如果你更喜欢XML的配置方式，'context:mbean-export'可以达到相同的目的。

```
<context:mbean-export/>
```

如果需要，你可以对特定的MBean 服务提供一个引用，并且defaultDomain（AnnotationMBeanExporter的属性）属性接受生成“ObjectNames”的备用值。如上节的MetadataNamingStrategy所述，它将用于代替完全限定的包名称。

```
@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDoma
in")
@Configuration
ContextConfiguration {

}
```

```
<context:mbean-export server="myMBeanServer" default-domain="myD
omain"/>
```

> 不要使用基于接口的AOP代理和JMX注解自动发现相结合的方式。基于接口的代理对隐藏目标类，它也会隐藏JMX管理的resource注解。因此，在这种情况下使用目标类代理：通过在, 等上设置'proxy-target-class'标志，否则，JMX也可能在启动时被忽略。

27.5 JSR-160连接器

对于远程访问，Spring JMX模块在org.springframework.jmx.support package包中提供了两种FactoryBean实现，用于创建服务端和客户端的连接器。

27.5.1 服务端连接器

为了使用Spring JMX 来创建，需要使用以下配置启动并暴露JSR-160 JMXConnectorServer：

```
<bean id="serverConnector" class="org.springframework.jmx.suppor
t.ConnectorServerFactoryBean"/>
```

ConnectorServerFactoryBean创建的JMXConnectorServer默认会绑定到"service:jmx:jmxmp://localhost:9875".serverConnector通过JMXMP协议在本地的9875端口上将本地的MBeanServer暴露给客户端。注意，JMXMP协议是JSR 160标记为可选协议：当前 ，JMX主要的开源实现MX4J，它只提供了基于JDK的协议，而不支持JMXMP。

分别使用serviceUrl和ObjectName属性来指明另一个URL并把JMXConnectorServer自身注册为一个MBeanServer：

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFa
ctoryBean">
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl"
            value="service:jmx:rmi://localhost/jndi/rmi://localh
ost:1099/myconnector"/>
</bean>
```

如果设置了ObjectName属性，Spring将自动使用在ObjectName底下使用MBeanServer注册连接器。下面的例子展示了当你创建一个JMXConnector时，你可以传递完整的参数给ConnectorServerFactoryBean。

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFa
ctoryBean">
    <property name="objectName" value="connector:name=iiop"/>
    <property name="serviceUrl"
        value="service:jmx:iiop://localhost/jndi/iiop://localhos
t:900/myconnector"/>
    <property name="threaded" value="true"/>
    <property name="daemon" value="true"/>
    <property name="environment">
        <map>
            <entry key="someKey" value="someValue"/>
        </map>
    </property>
</bean>
```

注意，当使用基于RMI连接器时，需要启动查找服务（tnameserv or rmiregistry）来完成名称注册。如果你使用Spring 通过RMI来导出远程服务，那么Spring已经构建了一个RMI注册。如果没有，你可以通过下面的配置简单的启动一个注册：

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiR
egistryFactoryBean">
    <property name="port" value="1099"/>
</bean>
```

27.5.2 客户端连接器

下面展示了使用MBeanServerConnectionFactoryBean创建远程JSR-160 MBeanServer 的MBeanServerConnection：

```
<bean id="clientConnector" class="org.springframework.jmx.suppor
t.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://localho
st/jndi/rmi://localhost:1099/jmxrmi"/>
</bean>
```

27.5.3 通过Hessian 或 SOAP 的JMX

JSR-160允许客户端和服务端之间进行通讯的方式进行扩展。上面的例子使用JSR-160规范（IIOP和JRMP）和（可选）JMXMP所需的强制基于RMI的实现。通过使用其他的提供者或JMX的实现（例如MX4J）你可以通过简单的HTTP或SSL或其他方式利用诸如SOAP或Hessian之类的协议：

```
<bean id="serverConnector" class="org.springframework.jmx.suppor
t.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=burlap"/>
    <property name="serviceUrl" value="service:jmx:burlap://loca
lhost:9874"/>
</bean>
```

上面的例子使用了 MX4J 3.0.0，关于MX4J请参考官方文档.

27.6 通过代理访问MBeans

Spring JMX 允许你创建代理，它将重新路由到本地或者远程MBeanServer中注册的MBean。这些代理提供了标准的Java接口来和MBean进行交互。下面的代码展示了如何在本地允许的MBeanServer中配置代理：

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

你可以看到在ObjectName: bean:name=testBean下注册的MBean创建代理。通过proxyInterfaces属性来控制代理实现的一系列接口，InterfaceBasedMBeanInfoAssembler使用和MBean上属性相同规则来操作这些接口上的方法映射规则和属性。

MBeanProxyFactoryBean可以通过MBeanServerConnection为任何可以访问的MBean创建一个代理。默认，使用本地的MBeanServer，但是你可以重写它，并提供一个指向远程MBeanServer的MBeanServerConnection来满足远程MBean的代理。

```
<bean id="clientConnector"
        class="org.springframework.jmx.support.MBeanServerConnec
tionFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://remoteh
ost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProx
yFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.j
mx.IJmxTestBean"/>
    <property name="server" ref="clientConnector"/>
</bean>
```

你可以看到，我们使用MBeanServerConnectionFactoryBean创建了一个指向远程机器的MBeanServerConnection。MBeanServerConnection通过server属性传递给MBeanProxyFactoryBean。创建的代理通过MBeanServerConnection将所有的调用都转发到MBeanServer。

27.7 通知

Spring JMX提供了对JMX通知的全面支持。

27.7.1 注册通知监听器

Spring JMX的支持使得将对任意数量的NotificationListeners注册到任意数量的MBean（包括通过Spring MBeanExporter 导出的MBean和通过其他机制注册的MBean）。示例，考虑这么一个场景，当目标MBean的每个属性每次改变的时候都会通知（通过通知）。

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
        implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Ob
ject handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notificati
on) {
        return AttributeChangeNotification.class.isAssignableFro
m(notification.getClass());
    }

}
```

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="test
Bean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="bean:name=testBean1">
                    <bean class="com.example.ConsoleLoggingNotif
icationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

通过上面的配置，目标MBean（bean:name=testBean1）每次以广播形式发送JMX通知，通过notificationListenerMappings属性注册为ConsoleLoggingNotificationListener的监听器将被通知。ConsoleLoggingNotificationListener可以采取任何它认为合适的动作来响应通知。

你可以直接使用bean的名称作为导出bena的监听器之间的连接：

```
<beans>

   <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="test
Bean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="testBean">
                    <bean class="com.example.ConsoleLoggingNotif
icationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

 </beans>
```

如果你想对封闭的MBeanExporter导出的所有bean注册一个NotificationListener实例，可以使用特殊通配符'*'（无引号）作为notificationListenerMappings属性map中的key；例如：

```
<property name="notificationListenerMappings">
    <map>
        <entry key="*">
            <bean class="com.example.ConsoleLoggingNotificationL
istener"/>
        </entry>
    </map>
</property>
```

如果需要执行反转（针对MBean注册一些不同的监听器），那么必须使用notificationListeners的属性列表（不是notificationListenerMappings属性）。这次，不是简单配置单个MBean的NotificationListener，而是配置NotificationListenerBean实例，NotificationListenerBean在MBeanServer中封装了NotificationListener和ObjectName（或ObjectNames）。NotificationListenerBean也封装了其他属性，例如NotificationFilter和用于高级JMX通知场景的任意handback对象。

使用NotificationListenerBean实例时和前面的不同配置：

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="test
Bean"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.Noti
ficationListenerBean">
                    <constructor-arg>
                        <bean class="com.example.ConsoleLoggingN
otificationListener"/>
                    </constructor-arg>
                    <property name="mappedObjectNames">
                        <list>
                            <value>bean:name=testBean1</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBe
an">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

上面的例子和第一个例子是等价的。现在假设我们想每发出一个通知就给出一个handback对象，除此之外我们还想通过一个NotificationFilter过滤外来的通知。（关于什么是handback对象，什么是NotificationFilter，请参考JMX规范（1.2）

的“JMX通知模型”）。

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MB
eanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="test
Bean1"/>
                <entry key="bean:name=testBean2" value-ref="test
Bean2"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.Noti
ficationListenerBean">
                    <constructor-arg ref="customerNotificationLi
stener"/>
                    <property name="mappedObjectNames">
                        <list>
                            <!-- handles notifications from two
distinct MBeans -->
                            <value>bean:name=testBean1</value>
                            <value>bean:name=testBean2</value>
                        </list>
                    </property>
                    <property name="handback">
                        <bean class="java.lang.String">
                            <constructor-arg value="This could b
e anything..."/>
                        </bean>
                    </property>
                    <property name="notificationFilter" ref="cus
tomerNotificationListener"/>
                </bean>
            </list>
        </property>
    </bean>
```

```
    <!-- implements both the NotificationListener and NotificationFilter interfaces -->
    <bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

    <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="ANOTHER TEST"/>
        <property name="age" value="200"/>
    </bean>

</beans>
```

27.7.2 发布通知

Spring不仅提供了对注册接受通知的支持，而且还用于发布通知。

> 请注意，本节仅与通过MBeanExporter暴露的Spring管理的MBean相关。任何现有的用户定义的MBean都应该使用标准的JMX API来发布通知。

Spring JMX支持的通知发布的关键接口为NotificationPublisher（定义在org.springframework.jmx.export.notification包下面）。任何通过MBeanExporter实例导出为MBean的bean都可以实现NotificationPublisherAware的相关接口来获取NotificationPublisher实例。NotificationPublisherAware接口通过一个简单的setter方法将NotificationPublisher的实例提供给实现bean，这个bean就可以用来发布通知。

如javadoc中的NotificationPublisher类所述，通过NotificationPublisher机制来发布事件被管理的bean是对任何通知监听器状态管理的不负责。Spring JMX支持将处理所有JMX基础问题。所有人需要做的就是和应用开发人员一样实现NotificationPublisherAware接口并通过NotificationPublisher实例开始发布事件。注意，NotificationPublisher将在管理bean被注册到MBeanServer之后被设置。

使用NotificationPublisher实例非常简单，创建一个简单的JMX通知实例（或一个适当的Notification子类实例），通知中包含发布事件相关的数据 ，然后在NotificationPublisher实例上调用sendNotification（Notification），传递Notification。

下面是一个简单的例子，在这种场景下，导出的JmxTestBean实例在每次调用add(int, int)时会发布一个NotificationEvent。

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationP
ublisherAware;
import org.springframework.jmx.export.notification.NotificationP
ublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPu
blisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add",
this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher n
otificationPublisher) {
        this.publisher = notificationPublisher;
    }

}
```

NotificationPublisher接口和使其全部运行的机制是Spring JMX支持的良好的功能之一。然而它带来的代价是你的类和Spring，JMX耦合在一起；与以往一样，我们给出实用的建议，如果你需要NotificationPublisher提供的功能，那么你需要接受

Spring和JMX的耦合。

27.8 更多资源

这章节包含了关于JMX更多的资源链接：

- Oracle的JMX主页
- JMX规范(JSR-000003)
- JMX远程API规范（JSR-000160）
- MX4J主页（JMX规范的各种开源实现）

29. 电子邮件

29.1 介绍

依赖库：使用Spring框架的邮件功能需要将JavaMail的Jar包添加到依赖中。这个库可以Maven中心找到：com.sun.mail:javax.mail。

Spring提供了一个实用的发送电子邮件库，它为使用者屏蔽了邮件系统的底层细节和客户端的底层资源处理。

Spring邮件相关功能在 `org.springframework.mail` 包下,其中 `MailSender` 是发送邮件的核心接口； `SimpleMailMessage` 类是对邮件属性（发件人、收件人以等）进行简单的封装。这个包中也包含一系列的检查异常，它们是对邮件系统低级别的异常进行抽象,且均继承自 `MailException` 。有关异常的更多信息,请参阅相关javadoc。

`org.springframework.mail.javamail.JavaMailSender` 接口继承自 `MailSender` 接口，并增加了一些特有的 `JavaMail` 功能，如*MIME*邮件的支持。 `JavaMailSender` 还提供了一个用于编写*MIME*消息的回调 `org.springframework.mail.javamail.MimeMessagePreparator` 接口。

29.2 使用

我们假设有一个 `OrderManager` 业务接口:

```
public interface OrderManager {

    void placeOrder(Order order);

}
```

假设我们需要生成一个有订单号的邮件，并发送给相关的客户。

29.2.1 `MailSender` 和 `SimpleMailMessage` 的基本用法

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class SimpleOrderManager implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage templateMessage;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setTemplateMessage(SimpleMailMessage templateMes
sage) {
        this.templateMessage = templateMessage;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        // Create a thread safe &amp;amp;quot;copy&amp;amp;quot;
 of the template message and customize it
        SimpleMailMessage msg = new SimpleMailMessage(this.templ
ateMessage);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            &amp;amp;quot;Dear &amp;amp;quot; + order.getCustome
r().getFirstName()
                + order.getCustomer().getLastName()
                + &amp;amp;quot;, thank you for placing order. Y
our order number is &amp;amp;quot;
                + order.getOrderNumber());
        try{
            this.mailSender.send(msg);
        }
```

```
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

在xml中添加相关的Bean定义：

```
<bean id="mailSender" class="org.springframework.mail.javamail.J
avaMailSenderImpl">
    <property name="host" value="mail.mycompany.com"/>
</bean>

<!-- this is a template message that we can pre-load with defaul
t state -->
<bean id="templateMessage" class="org.springframework.mail.Simpl
eMailMessage">
    <property name="from" value="customerservice@mycompany.com"/
>
    <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support
.SimpleOrderManager">
    <property name="mailSender" ref="mailSender"/>
    <property name="templateMessage" ref="templateMessage"/>
</bean>
```

29.2.2 使用 `JavaMailSender` 和 `MimeMessagePreparator`

下面的例子是 `OrderManager` 接口的另一种实现，其中使用了 `MimeMessagePreparator` 类。在这里， `mailSender` 是 `JavaMailSender` 类型，因此我们可以使用 `JavaMail MimeMessage` 类：

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePrepar
ator() {

            public void prepare(MimeMessage mimeMessage) throws
Exception {

                mimeMessage.setRecipient(Message.RecipientType.T
O,
                        new InternetAddress(order.getCustomer().
getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress(&amp;amp
;quot;mail@mycompany.com&amp;amp;quot;));
                mimeMessage.setText(
                        &amp;amp;quot;Dear &amp;amp;quot; + orde
r.getCustomer().getFirstName() + &amp;amp;quot; &amp;amp;quot;
                            + order.getCustomer().getLastName()
```

```
                            + &amp;amp;quot;, thank you for plac
ing order. Your order number is &amp;amp;quot;
                            + order.getOrderNumber());
            }
        };

        try {
            this.mailSender.send(preparator);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

> 上面中邮件代码只是作为示例，最好方式是将邮件发送代码重构到其它Bean中，并在OrderManager合适的地方调用它。

Spring框架邮件也支持标准的JavaMail实现。了解更多信息，请参阅相关的javadocs。

29.2 使用MimeMessageHelper

`org.springframework.mail.javamail.MimeMessageHelper` 是一个处理JavaMail消息的好工具，它屏蔽了很多JavaMail API的细节，所以使用 `MimeMessageHelper` 可以很简便的创建一个 `MimeMessage` 。

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost(&amp;amp;quot;mail.host.com&amp;amp;quot;);

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo(&amp;amp;quot;test@host.com&amp;amp;quot;);
helper.setText(&amp;amp;quot;Thank you for ordering!&amp;amp;quo
t;);

sender.send(message);
```

29.3.1 附件和嵌入资源

邮件允许添加附件和内联资源。嵌入资源是你嵌入到邮件中的图片或样式，但又不希望显示为附件。

附件

下面的例子将展示如何使用 `MimeMessageHelper` 发送一个带JPEG图片附件的邮件：

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost(&amp;amp;quot;mail.host.com&amp;amp;quot;);

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo(&amp;amp;quot;test@host.com&amp;amp;quot;);

helper.setText(&amp;amp;quot;Check out this image!&amp;amp;quot;
);

// let's attach the infamous windows Sample file (this time copi
ed to c:/)
FileSystemResource file = new FileSystemResource(new File(&amp;a
mp;quot;c:/Sample.jpg&amp;amp;quot;));
helper.addAttachment(&amp;amp;quot;CoolImage.jpg&amp;amp;quot;,
file);

sender.send(message);
```

嵌入资源

下面的例子将展示如何使用 `MimeMessageHelper` 发送一个嵌入图片的邮件：

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost(&amp;amp;quot;mail.host.com&amp;amp;quot;);

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo(&amp;amp;quot;test@host.com&amp;amp;quot;);

// use the true flag to indicate the text included is HTML
helper.setText(&amp;amp;quot;&amp;amp;lt;html&amp;amp;gt;&amp;amp;lt;body&amp;amp;gt;&amp;amp;lt;img src='cid:identifier1234'&amp;amp;gt;&amp;amp;lt;/body&amp;amp;gt;&amp;amp;lt;/html&amp;amp;gt;&amp;amp;quot;, true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File(&amp;amp;quot;c:/Sample.jpg&amp;amp;quot;));
helper.addInline(&amp;amp;quot;identifier1234&amp;amp;quot;, res);

sender.send(message);
```

> 嵌入资源需要使用Content-ID(上面的例子identifier1234)添加到MIME消息中。文本和嵌入资源添加是有顺序的，需要按照先添加文本，再添加嵌入资源的顺序。否则，它将不会工作!

29.3.2 使用模板库创建电子邮件内容

在前面的例子中，我们通常使用 `message.setText(..)` 等方法创建邮件内容。在简单的情况下，像前面例子那样使用API就可以满足我们的需要了。

在典型的企业应用程序中,下面的原因让你不定会使用上面的方法创建你的邮件内容。

- 在Java代码中创建HTML的电子邮件内容冗长，且容易出错
- 呈现逻辑和业务逻辑混杂

- 更改电子邮件内容的展示结构需要编写Java代码，重新编译，重新部署...

通常解决方法是使用模板框架定义电子邮件的呈现逻辑,如FreeMarker。分离呈现逻辑和业务逻辑使得你的代码更清晰。当你的邮件的内容变的复杂时，这绝对是一个最佳实践，而且Spring框架对FreeMarker有很好的支持。

32. 缓存

32.1 介绍

Spring 框架从 3.1 开始，对 Spring 应用程序提供了透明式添加缓存的支持。和事务支持一样，抽象缓存允许一致地使用各种缓存解决方案，并对代码的影响最小。

从 4.1 版本开始，缓存抽象支持了 JSR-107 注释和更多自定义选项，从而得到了显著的改进。

32.2 缓存抽象

```
缓存（Cache） vs 缓冲区（Buffer）

缓存和缓冲区两个术语往往可以互换着使用。但注意，它们代表着不同的东西。
缓冲区是作用于快和慢速实体之间的中间临时存储。
一块缓冲区必须等待其他并影响性能，通过允许一次性移动整个数据块而不是小块来缓
解。数据从缓冲区读写只有一次。因此缓冲区对至少一方是可见的。
另一方面，缓存根据定义是隐性的，双方不会知道缓存的发生。它提高了性能，但允许
以快速的方式多次读取相同的数据。

想了解更多这二者之间的差异，见：https://en.wikipedia.org/wiki/Cache_(
computing)#The_difference_between_buffer_and_cache
```

核心上，抽象将缓存作用于 Java 方法上，基于缓存中的可用信息，可以减少方法的执行次数。也就是说，每次目标方法的调用时，抽象使用缓存行为来检查执行方法，检查执行方法是否给定了缓存的执行参数：如果有，则返回缓存结果，不执行具体方法；如果没有，则执行方法，并将结果缓存后，返回给用户。以便于下次调用方法时，直接返回缓存的结果。这样，只要给定缓存执行参数，在复杂的方法（无论是 CPU 或者 IO 相关）只需要执行一次，就可以得到结果，并利用缓存可重复使用结果，而不必再次执行该方法。另外，缓存逻辑可以被透明地调用，不会对调用者造成任何的困扰。

> 显然，这种方法只适用于为某个给定输入（或参数）返回相同输出（结果），无论执行多少次。

抽象提供的其他缓存相关操作，比如更新缓存内容或者删除其中一条缓存。如果在应用程序过程中，发生了变化的数据需要缓存，那这些功能会很有用。

比如 Spring 框架其他服务一样，缓存服务是一种抽象（不是缓存的实现），并且需要使用实际的存储器来存储缓存数据。也就是说，抽象能够使开发人员不必编写缓存逻辑，但它没有提供缓存的存储器。这个抽象是由 `org.springframework.cache.Cache` 和 `org.springframework.cache.CacheManager` 接口实现的。

有些抽象的实现是开箱即用的：基于 JDK `java.util.concurrent.ConcurrentMap` 缓存实现，Ehcache 2.x，Gemfire cache, Caffeine 和 JSR-107 缓存（例如 Ehcache 3.x）。有关缓存存储/提供的更多信息，请参见 32.7 节 《Plugging-in different back-end caches》。

> 缓存抽象没有特别处理多线程和多进程环境，因为这些功能由缓存实现来处理...

如果是多进程环境（即部署在多个节点上的应用程序），则需要相应的配置程序提供缓存。根据使用情况，几个节点上相同数据的副本可能足够多了，但如果在应用程序过程中更改了数据，则需要启动其他传播机制，进行同步缓存数据。

缓存一个特定的对象是典型的缓存交互 get-if-not-found-then-proceed-and-put-finally 代码块：不需应用任何锁，并且几个线程同时尝试加载相同的对象。同样适用于回收，如果多个线程同时更新或者回收数据，则可能会使用过时的数据。某些缓存提供者在该领域提供高级功能，请参考您正在使用的缓存提供者的更多高级功能的详细信息。

要使用缓存抽象，开发人员需要注意两个方面：

- 缓存声明 - 标志缓存的方法及缓存策略
- 缓存配置 - 数据读写的缓存数据库

32.3 基于声明式注解的缓存

对于缓存声明，抽象提供了一组 Java 注解：

- `@Cacheable` 触发缓存机制
- `@CacheEvict` 触发缓存回收
- `@CachePut` 更新缓存，而不会影响方法的执行
- `@Caching` 组合多个缓存操作到一个方法
- `@CacheConfig` 类级别共享系诶常见的缓存相关配置

下面，让我们仔细看看每个注释。

32.3.1 @Cacheable 注解

顾名思义，`@Cacheable` 用于标识可缓存的方法 - 即需要将结果存储到缓存中的方法，以便于再一次调用（具有相同的输入参数）时返回缓存结果，而无需执行该方法。在最简单的形式中，注解声明需要定义与注解方法相关联的缓存名称：

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

在上面的代码中，`findBook` 与名为 `books` 的缓存相关。每次调用该方法时，都会检查缓存以查看调用是否已经被执行，并且不必重复。而在大多数情况下，只有一个缓存被声明，注解允许指定多个名称，以便使用多个缓存。在这种情况下，执行该方法之前将检查每个高速缓存 - 如果至少有一个缓存被命中，则返回缓存相关的值：

> 注意：即使没有实际执行缓存方法，所有其他不包含该值的缓存也将被更新。

```
@Cacheable({"books", "isbns"})
public Book findBook(ISBN isbn) {...}
```

默认键生成

缓存的本质是键值对存储，所以每次调用缓存方法都会转换作用于缓存访问合适的键。开箱即用，缓存抽象使用基于以下算法的简单 `KeyGenerator`：

- 如果没有参数，返回 `SimpleKey.EMPTY`
- 如果只有一个参数，返回该实例
- 如果大于一个参数，返回一个包含所有参数的 `SimpleKey`

这种算法对大多数用例很适用，只要参数具有自然键并实现了有效的 `hashCode()` 和 `equals()` 方法。如果不是这样，策略就需要改变。要提供不同的默认密钥生成器，需要实现 `org.springframework.cache.interceptor.KeyGenerator` 接口。

> Spring 4.0 的发布，默认键生成策略发生了变化。Spring 早期版本使用的键生成策略对于多个关键参数，只考虑了参数的 `hashCode()` ，而没有考虑 `equals()` 。这可能会导键碰撞（参见 SPR-10237 资料）。新的 `SimpleKeyGenerator` 对这种场景使用了复合键。
> 如果要继续使用以前的键策略，可以配置不推荐使用的 `org.springframework.cache.interceptor.DefaultKeyGenerator` 类或者创建基于哈希的自定义 `KeyGenerator` 的实现

自定义键生成声明

缓存是通用的，因此目标方法可能有不能简单映射到缓存结构之上的签名。当目标方法具有多个参数时，这一点往往变得很明显，其中只有一些参数适合于缓存（其他的仅由方法逻辑使用）。例如：

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)
```

第一眼看代码，虽然两个 `boolean` 参数影响了该 `findBook` 方法，但对缓存没有任何用处。更进一步，如果两者之中只有一个是重要的，另一个不重要呢？

这种情况下， `@Cacheable` 注解只允许用户通过键属性指定 `key` 的生成方式。开发人员可以使用 SpEL 来选择需要的参数（或其嵌套属性），执行参数设置调用任何方法，无需编写任何代码或者实现人任何接口。这是默认键生成器推荐的方法，因为方法在代码库的增长下，会有完全不同的方法实现。而默认策略可能适用于某些方法，并不是适用于所有的方法。

这里是 SpEL 声明的一些示例 - 如果你不熟悉它，查阅Chapter 6, Spring Expression Language (SpEL)：

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)

@Cacheable(cacheNames="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)

@Cacheable(cacheNames="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)
```

上面代码片段显示了选择某个参数，或参数的某个属性值或任意（静态）方法，如此方便操作。

如果生成键的算法太具体或者需要共享，可以操作中定义一个自定义的 `keyGenerator` 。为此，请指定要使用的 `KeyGenerator` Bean 实现的名称：

```
@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)
```

> key 和 keyGenerator 的参数是互斥的，指定两者的同样的操作将导致异常。

默认缓存解析

开箱即用，缓存抽象使用一个简单的 `CacheResolver` ,在 `CacheManager` 可以配置操作级别来检索缓存。

需要不同的默认缓存解析器，需要实现接口 `org.springframework.cache.interceptor.CacheResolver` 。

自定义缓存解析

默认缓存解析适用于使用单个 `CacheManager` 并且应用在不需要复杂缓存解析的应用程序。

对于使用多个缓存管理器的应用，可以为每个操作设置一个 `cacheManager`：

```
@Cacheable(cacheNames="books", cacheManager="anotherCacheManager
")
public Book findBook(ISBN isbn) {...}
```

也可以完全以与键生成类似的方式来替换 `CacheResolver` 。每个缓存操作都要求缓存解析，基于运行时参数的缓存解析:

```
@Cacheable(cacheResolver="runtimeCacheResolver")
public Book findBook(ISBN isbn) {...}
```

> 自 Spring 4.1 以后，缓存注解的属性值是不必要的，因为 CacheResolver 可以提供该特定的信息，无论注解的内容是什么。与 key 和 keyGenerator 类似，cacheManager 和 cacheResolver 参数是互斥的，并且指定两者同样的操作会导致异常，因为 CacheResolver 的实现将忽略自定义的 CacheManager。这是你不希望的。

同步缓存

在多线程环境中，某些操作可能会导致同时引用相同的参数（通常在启动时）。默认情况下，缓存抽象不会锁定任何对象，同样的对象可能会被计算好几次，从而达不到缓存的目的。

对于这些特熟情况， `sync` 属性可用于指示底层缓存提供程序在计算该值时锁定缓存对象。因此，只有一个线程将忙于计算值，而其他线程会被阻塞，直到该缓存对象被更新为止。

```
@Cacheable(cacheNames="foos", sync="true")
public Foo executeExpensiveOperation(String id) {...}
```

> 这是可选功能，可能你是用的缓存库不支持它。由核心框架提供的所有 CacheManager 都会实现并支持它。翻阅缓存提供商文档可以了解更多详细信息。

条件缓存

有时，一种方法可能不适合缓存（例如，它可能取决于给定的参数）。缓存注解通过条件参数支持这样的功能，采用使用 `SpEL` 表达式的 `condition` 参数表示。如果是 true，则缓存方法，如果是 false，则不缓存，无论缓存中有什么值或者使用了哪些参数都严格按照规则执行该方法。一个快速入门的例子 - 只有当参数名称长度小于 32 的时候，才会缓存下面方法：

```
@Cacheable(cacheNames="book", condition="#name.length < 32")
public Book findBook(String name)
```

另外，`unless` 参数用于是否向缓存添加值。不同于 `condition` 参数，`unless` 参数在方法被调用后评估表达式。扩展上一个例子 - 我们只想缓存平装书：

```
@Cacheable(cacheNames="book", condition="#name.length < 32", unless="#result.hardback")
public Book findBook(String name)
```

缓存抽象支持 `java.util.Optional`,只有当它作为缓存值的时候才可使用。`#result` 指向的是业务实体，不是在包装类上。上面的例子可以重写如下：

```
@Cacheable(cacheNames="book", condition="#name.length < 32", unless="#result.hardback")
public Optional<Book> findBook(String name)
```

注意：结果仍然是 `Book` 不是 `Optional`

缓存 **SpEL** 上下文

每个 SpEL 表达式都会有求值上下文。除了构建参数，框架提供了专门的缓存相关元数据，比如参数名称。下表列出了可用于上下文的项目，因此可以使用他们进行键和条件的计算：

表 32.1 缓存 SpEL 元数据

参数名	用处	描述	例子
methodName	root object	被调用的方法名	`#root.methodName`
method	root object	被调用的方法	`#root.method.name`
target	root object	被调用的对象	`#root.target`
targetClass	root object	被调用的类	`#root.targetClass`
args	root object	被调用的的类目标参数	`#root.args[0]`
caches	root object	当前方法执行的缓存列表	`#root.caches[0].name`
argument name	evaluation context	任何方法参数的名称	`#iban` 或 `#a0` (也可以使用 `#p0` 或者 `#p<#arg>`)
result	evaluation context	方法调用的结果（缓存的值）	`#result`

32.3.2 @CachePut 注解

需要缓存更新但不影响方法执行的情况，可以使用 `@CachePut` 注解。也就是说，该方法始终执行，并将其结果放入缓存中（根据 `@CachePut` 选项）。它支持与 `@Cacheable` 相同的选项，适用于缓存而不是方法流程优化：

```
@CachePut(cacheNames="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

> 对同一个方法同时使用 `@CachePut` 和 `@Cacheable` 注解通常是不推荐的。因为它们有不同的行为。后者通过使用缓存并跳过方法执行，前者强制执行方法并进行缓存更新。这会导致意想不到的行为，除了具体的场景（例如需要排除条件的注解），应该避免这样的声明方式。还要注意，这样的条件不应该依赖于结果对象（#result 对象），因为结果对象应该在前面被验证后排除。

32.3.3 @CacheEvict 注解

缓存抽象不仅仅缓存更多数据，还可以回收缓存。这个过程对于从缓存中删除旧数据或者未使用的数据非常有用。与 `@Cacheable` 相反，注解 `@CacheEvict` 划分了回收缓存的方法，即作为从缓存中删除数据的触发器方法。`@CacheEvict` 需要指定一个（或者多个）执行动作，并且允许自定义缓存和键解析或条件被指定。但有一个额外的参数 `allEntries`,可以指示是否所有对象都要被收回，还是一个对象（取决于key）。

```
@CacheEvict(cacheNames="books", allEntries=true)
public void loadBooks(InputStream batch)
```

当整个缓存区需要被收回时，这个选项就会派上用场 - 而不是回收每个对象（当不起作用时会需要很长的响应时间），所以对象会在一个操作中被删除，如上所示。注意，框架将忽略此场景下指定的任何key，因为它不适用（整个缓存收回，不仅仅是一个条目被收回）。

还可以指出发生缓存回收是在默认之后还是在通过 `beforeInvocation` 属性执行方法后。前者提供与其他注解相同的语义 - 一旦方法成功完成，就执行缓存上的动作（这种情况是回收）。如果方法不执行（因为它可能会被缓存）或者抛出异常，则不会发生回收。后者（ `beforeInvocation = ture` ）会导致在调用该方法之前发生回收 - 在驱逐不需要与方法结果相关的情况下，这是有用的。

重要的是， `void` 方法可以和 `@CacheEvict` 一起使用 - 由于方法作为触发器，返回值会被忽略（因为他们不与缓存交互） - `@Cacheable` 就不是这样的，它添加 / 更新数据到缓存时，需要一个结果。

32.3.4 @Caching 注解

另外情况下，需要指定想同类型的多个注解，例如 `@CacheEvict` 或 `@CachePut` 需要被指定。比如因为不同缓存之间的条件或者键表达式不同。@Caching 允许在同一个方法上使用多个嵌套的 `@Cacheable` 、 `@CachePut` 和 `@CacheEvict` ：

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheName
s="secondary", key="#p0") })
public Book importBooks(String deposit, Date date)
```

32.3.5 @CacheConfig 注解

到目前为止，我们已经看到缓存操作提供了许多定制选项，这些选项可以在操作的基础上进行设置。 但是，一些自定义选项可能很麻烦，可以配置是否适用于该类的所有操作。 例如，指定用于该类的每个缓存操作的高速缓存的名称可以被单个类级别定义所替代。 这是`@CacheConfig`发挥作用的地方。

```
@CacheConfig("books")
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) {...}
}
```

`@CacheConfig` 是一个类级别的注解，可以共享缓存名称。自定义 `KeyGenerator` ，自定义 `CacheManager` 和自定义 `CacheResolver` 。该注解在类上不会操作任何缓存。

操作级别上的自定义会覆盖 `@CacheConfig` 的自定义。因此，每个缓存操作都会有三个级别的定义：

- 全局配置，可用于 `CacheManager` 和 `KeyGenerator`
- 类级别，用 `@CacheConfig`
- 操作级别层面

@EnableCaching 注解

重点注意的是，即使声明缓存注解也不会主动触发动作 - Spring 中很多东西都这样，该特性必须声明式的启用（意味着如果缓存出现问题，则通过删除某一个配置就可以验证，而不是所有代码的注释）。

启动缓存注解，将 `@EnableCaching` 注解加入到 `@Configuration` 类中：

```
@Configuration
@EnableCaching
public class AppConfig {
}
```

对于 XML 配置，使用 `cache:annotation-driven` 元素：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.s
pringframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache http://www.s
pringframework.org/schema/cache/spring-cache.xsd">

        <cache:annotation-driven />

</beans>
```

`cache:annotation-driven` 元素和 `@EnableCaching` 注解允许指定多种选项，这些选项通过 AOP 将缓存行为添加到应用程序的方式。该配置与 `@Transactional` 注解类似：

> Java 高级自定义配置需要实现 `CachingConfigurer` 。更多详细信息，参考 javadoc。

表 32.2. 缓存注解设置

XML 属性	注解属性	默认	
	N/A（参阅		要使用缓存管

cache-manager	N/A（参阅 CachingConfigurer javadocs）	cacheManager	认 `Cac` 理缓存
cache-resolver	N/A（参阅 CachingConfigurer javadocs）	配置 `cacheManager` 的 `SimpleCacheResolver`	Cache 不是 m
key-generator	N/A（参阅 CachingConfigurer javadocs）	`SimpleKeyGenerator`	要使
error-handler	N/A（参阅 CachingConfigurer javadocs）	`SimpleCacheErrorHandler`	要使用 名称。 期间
mode	mode	proxy	默认 架处 义，如 式“asp 缓存方 类字节 用。 spring 织（或 置加 t
proxy-target-class	proxyTargetClass	false	仅 用 `@Ca` 释注释 果 `pr` 为 `tr` `class` 省略， 接口的 理 “
order	order	Ordered.LOWEST_PRECEDENCE	用 `@Ca` 释的b 关与A 多信

				序意

> `<cache：annotation-driven />` 只会匹配相对应的应用上下文中定义的 `@Cacheable` / `@CachePut` / `@CacheEvict` / `@Cached` 。如果将 `<cache:annotation-driven/>` 配置在 `DispatcherServlet` 的 `WebApplicationContext` ，它只检查控制器中的bean，而不是您的服务。参考 18.2 章节 `DispatcherServlet` 获取更多信息。

方法可见性和缓存注解
使用代理后，缓存注解应用于公共可见的方法。如果对 protected、private 或 package-visible 方法进行缓存注解，不会引起错误。但注解的方法不会显示缓存配置。如果更改字节码时需要注解非公共方法，请考虑使用 AspectJ（见下文）。

> Spring 建议只使用 `@Cache*` 注解具体的类（或具体的方法），而不是注解接口。可以在接口（或接口方法）上注解 `@Cache*` ，但只有当使用基于接口的代理时，才能使用它。Java 注解不是用接口继承，如果使用基于类的代理（ `proxy-target-class="true"` ）或基于代理的 `weaving-based(mode="aspectj")` ，缓存设置无法识别，对象也不会被包装在缓存代理中。
>
> 在代理模式（默认情况）中，只有通过代理进入的外部方法调用被截取。实际上，自调用目标对象中调用另一个目标对象方法的方法在运行时不会导致实际的缓存，即使被调用的方法被 `@Cacheable` 标志 - 可以考虑 aspectj 模式。此外，代理必须被完全初始化提供预期行为，因此不应该依赖于初始化的代码功能，即 `@PostConstruct`

32.3.7 使用自定义注解

自定义注解和 AspectJ
开箱即用，此功能仅使用基于代理的方法，但可使用 AspectJ 进行一些额外的工作。

spring-aspects 模块仅为标准注解定义了一个切面。如果你定义了自己的注解，那还需要为这些注解定义一个方面。检查 AnnotationCacheAspect 为例。

缓存抽象允许使用不同的注解识别什么方法触发缓存或者缓存回收。这是非常方便的模板机制，因为不需要重复缓存声明（特别是在指定键和条件），或者在代码库不允许使用外部的导入（ `org.springframework` ）。其他的注解 @Cacheable, @CachePut, @CacheEvict 和 @CacheConfig 可作为元注解，也可以被其他注解使用。换句话说，可以自定义一个通用的注解替换 `@Cacheable` 声明：

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(cacheNames="books", key="#isbn")
public @interface SlowService {
}
```

以上我们定义的注解 `@SlowService` ，并使用 `@Cacheable` 注解 - 现在我们替换下面的代码：

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)
```

改为：

```
@SlowService
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)
```

即使 `@SlowService` 不是Spring注解，容器也可以在运行时自动选择其声明并了解其含义。 请注意，如上所述，需要启用注解驱动的行为。

32.4 JCache (JSR-107) 注解

Spring Framework 4.1 以来，缓存抽象完全支持 JCache 标准：即 `@CacheResult` ，`@CachePut` ，`@CacheRemove` 和 `@CacheRemoveAll` 还有 `@CacheDefaults` ，`@CacheKey` 和 `@CacheValue` 。这些注解被大家正确的使用，体现了缓存在 `JSR-107` 的实现：缓存抽象的内部实现，并提供了符合规范的默认 `CacheResolver` 和 `KeyGenerator` 的实现。换句话说，如果你已经使用了 Spring 缓存抽象，那可以平滑切换到这些标准注解，无需更改缓存存储（或者配置）。

32.4.1 特征总结

对于熟悉 Spring 缓存注解，下面描述了 Spring 注解和 JSR-107 对应的主要区别：

表 32.3. Spring vs JSR-107 缓存注解

Spring	JSR-107	备注
`@Cacheable`	`@CacheResult`	相似， `@CacheResu` 可缓存特定的异 并强制执行该方 不管缓存的结果
`@CachePut`	`@CachePut`	Spring 调用方法 获取结果去更新 存，但 JCache 作为使用 `@CacheValue` 数传递。所以 JCache 允许实 法调用之前或者 更新缓存。
`@CacheEvict`	`@CacheRemove`	相似， `@CacheRemove` 持条件回收，以 方法调用导致异
`@CacheEvict(allEntries=true)`	`@CacheRemoveAll`	查阅 `@CacheRemov`
`@CacheConfig`	`@CacheDefaults`	允许类似的方式 相同的属性

JCache 的 `javax.cache.annotation.CacheResolver` 概念和 Spring 的 `CacheResolver` 是一样的，除了 JCache 只支持单个缓存。默认下，根据注解声明的名称检索要使用的缓存。如果没有指定缓存的名称，则会自动生成默认值。更多信息可以查看 javadoc 的 `@CacheResult#cacheName()` 。

`CacheResolverFactory` 检索出 `CacheResolver` 实例。每个缓存操作都可以自定义工厂：

```
@CacheResult(cacheNames="books", cacheResolverFactory=MyCacheRes
olverFactory.class)
public Book findBook(ISBN isbn)
```

> 注意
> 对于所有引用的类，Spring 会找到具体指定类型的 Bean。如果存在多个匹配，会创建一个新实例，并可以正常的 Bean 生命周期回调（如依赖注入）

键由 javax.cache.annotation.CacheKeyGenerator 生成，和 Spring 的 KeyGenerator 能达到相同的目标。默认情况下，所有方法参数都会被考虑，除非至少有一个参数被注解为 @CacheKey。这和 Spring 的自定义键生成类似。例如，下面代码操作是相同的，一个使用 Spring 抽象，另一个使用 JCache：

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean
includeUsed)

@CacheResult(cacheName="books")
public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse
, boolean includeUsed)
```

CacheKeyResolver 可以在操作中指定，类似方式是 CacheResolverFactory。

JCache 管理注解方法抛出的异常：可以防止缓存更新，但将异常作为故障的标志，而不是再次调用该方法。假设抛出 InvalidIsbnNotFoundException 异常，那么 ISBN 的结构是无效的。如果这是一个永久的失败，没有任何书会被查询出来。以下的缓存异常，以使具有无效的 ISBN 进一步直接抛出缓存的异常，而不是再次调用该方法。

```
@CacheResult(cacheName="books", exceptionCacheName="failures"
             cachedExceptions = InvalidIsbnNotFoundException.cla
ss)
public Book findBook(ISBN isbn)
```

32.4.2 启用 JSR-107 支持

不需要特殊处理，去支持 JSR-107 和 Spring 的声明式注解。JSR-17 API 和 `spring-context-support` 模块在类路径下，`@EnableCaching` 和 `cache:annotation-driven` 两者会被启用。

> 根据你的具体案例选择需要的。你还可以使用 JSR-107 API 和其他使用 Spring 自己的注解来匹配服务。注意，如果这些服务影响相同的缓存，则使用一致的键生成实现。

32.5 缓存声明式 XML 配置

如果不想使用注解，可以使用 XML 进行声明式配置缓存。所以不用注解方法的形式，而从外部指定目标方法和缓存指令（类似于声明式事务管理）。以前的例子可以转化为：

```
<!-- the service we want to make cacheable -->
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#isbn"/>
        <cache:cache-evict method="loadBooks" all-entries="true"
/>
    </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces
-->
<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(*
x.y.BookService.*(..))"/>
</aop:config>

<!-- cache manager definition omitted -->
```

上面的配置中，`bookService` 是可配缓存的服务。在 `cache:advice` 指定方法 `findBooks`

32.6 配置缓存存储

开箱即用，缓存抽象提供了多种存储集成。要使用它们，需要简单地声明一个适当的 `CacheManager` - 一个控制和管理 `Cache` s，可用于检索这些存储。

32.6.1 JDK ConcurrentMap-based Cache

基于JDK的 `Cache` 实现位于 `org.springframework.cache.concurrent` 包下。它允许使用 `ConcurrentHashMap` 作为后备缓存存储。

```
<!-- simple cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support
.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean class="org.springframework.cache.concurrent.Co
ncurrentMapCacheFactoryBean" p:name="default"/>
            <bean class="org.springframework.cache.concurrent.Co
ncurrentMapCacheFactoryBean" p:name="books"/>
        </set>
    </property>
</bean>
```

上面的代码片段使用 `SimpleCacheManager` 创建一个 `CacheManager` ,为两个嵌套的 `ConcurrentMapCache` 实例命名为*default*和*books*。请注意，这些名称是为每个缓存直接配置的。

由于缓存是由应用程序创建的，因此它必须与其生命周期一致，使其适用于基本用例，测试或简单应用程序。缓存规模好，速度快，但不提供任何管理、持久化能力或驱逐合同。

32.6.2 Ehcache-based Cache（基于Ehcache的缓存）

> Ehcache 3.x完全符合JSR-107标准，不需要专门的支持。

Ehcache 2.x的实现位于org.springframework.cache.ehcache包下。要使用它，只需要声明适当的 `CacheManager` ：

```
<bean id="cacheManager"
      class="org.springframework.cache.ehcache.EhCacheCacheManag
er" p:cache-manager-ref="ehcache"/>

<!-- EhCache library setup -->
<bean id="ehcache"
      class="org.springframework.cache.ehcache.EhCacheManagerFac
toryBean" p:config-location="ehcache.xml"/>
```

此设置引导Spring IoC（通过ehcache bean）中的 `ehcache` 库，然后将其连接到专用的 `CacheManager` 实现中。请注意，整个ehcache特定的配置是从 `ehcache.xml` 读取的。

32.6.3 Caffeine Cache

Caffeine是Java 8的重写Guava缓存，其实现位于 `org.springframework.cache.caffeine` 包下，并提供了对Caffeine的几项功能的访问。

根据需要，配置创建缓存的 `CacheManager` 很简单：

```
<bean id="cacheManager"
      class="org.springframework.cache.caffeine.CaffeineCacheMan
ager"/>
```

还可以明确提供使用的缓存。在这种情况下，只有manager才能提供：

```
<bean id="cacheManager" class="org.springframework.cache.caffein
e.CaffeineCacheManager">
    <property name="caches">
        <set>
            <value>default</value>
            <value>books</value>
        </set>
    </property>
</bean>
```

Caffeine `CacheManager` 也支持自定义的 `Caffeine` 和 `CacheLoader` 。查看Caffeine documentation获取更多信息。

32.6.4 GemFire-based Cache（基于GemFire的缓存）

GemFire是面向内存/磁盘支持，弹性可扩展，持续可用，主动（内置基于模式的订阅通知），全局复制数据库，并提供功能齐全的边缘缓存。有关如何使用GemFire作为CacheManager（及更多）的更多信息，请参考 Spring Data GemFire reference documentation。

32.6.5 JSR-107 Cache

Spring的缓存抽象也可以使用兼容JSR-107的缓存。JCache实现位于 `org.springframework.cache.jcache` 包下。
要使用它，只需要声明适当的 `CacheManager` ：

```
<bean id="cacheManager"
      class="org.springframework.cache.jcache.JCacheCacheManager
"
      p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup  -->
<bean id="jCacheManager" .../>
```

32.6.6 Dealing with caches without a backing store（处理没有后端存储的缓存）

有时在切换环境或进行测试时，可能会有缓存声明，而不配置实际的后备缓存。由于这是一个无效的配置，因此在运行时将会抛出异常，因为缓存基础结构无法找到合适的存储。在这种情况下，而不是删除缓存声明（这可以证明是乏味的），可以连接一个不执行缓存的简单的虚拟缓存，也就是强制每次执行缓存的方法：

```
<bean id="cacheManager" class="org.springframework.cache.support
.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache"/>
            <ref bean="gemfireCache"/>
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

上面的 `CompositeCacheManager` 链接多个 `CacheManager` s，另外，通过 `fallbackToNoOpCache` 标志，添加了一个*no op*缓存，用于所有不被配置的缓存管理器处理的定义。也就是说，在 `jdkCache` 或 `gemfireCache` （上面配置）中找不到的每个缓存定义都将由*no op*缓存来处理，这不会存储任何导致目标方法每次执行的信息。

32.7 插入不同的后端缓存

显然，有很多缓存产品可以用作后备存储。要插入它们，需要提供一个 `CacheManager` 和 `Cache` 实现，因为不幸的是没有可用的标准供我们使用。这听起来比实际上更难听，这些类往往是简单的适配器，将缓存抽象框架映射到存储API的顶部，就像ehcache类可以显示一样。大多数 `CacheManager` 类都可以使用 `org.springframework.cache.support` 包中的类，例如 `AbstractCacheManager` ，其中只需要完成实际的映射即可完成代码。我们希望及时提供与Spring集成的库可以填补这个小的配置差距。

32.8 如何设置 TTL/TTI/Eviction policy/XXX 功能?

直接通过您的缓存提供商。缓存抽象是一个抽象而不是缓存实现。您正在使用的解决方案可能支持各种数据策略和其他解决方案不同的拓扑（例如JDK `ConcurrentHashMap`），因为缓存中的提取将无济于事，因为不需要后台支持。这样的功能应该通过后台缓存，配置它或通过其本机API直接控制。

Spring框架的新功能

这一章主要提供Spring框架新的功能和变更。

升级到新版本的框架可以参考。Spring git。

内容列表

Spring FrameWork 5.0新的功能

JDK 8+和Java EE7+以上版本

整个框架的代码基于java8

- 通过使用泛型等特性提高可读性
- 对java8提高直接的代码支撑
- 运行时兼容JDK9
- Java EE 7API需要Spring相关的模块支持
- 运行时兼容Java EE8 API
- 取消的包,类和方法
- 包 beans.factory.access
- 包 dbc.support.nativejdbc
- 从spring-aspects 模块移除了包mock.staicmock,不在提 AnnotationDrivenStaticEntityMockingControl支持
- 许多不建议使用的类和方法在代码库中删除

核心特性

JDK8的增强：

- 访问Resuouce时提供getFile或和isFile防御式抽象

- 有效的方法参数访问基于java 8反射增强
- 在Spring核心接口中增加了声明default方法的支持一贯使用JDK7 Charset和StandardCharsets的增强
- 兼容JDK9
- Spring 5.0框架自带了通用的日志封装
- 持续实例化via构造函数(修改了异常处理)
- Spring 5.0框架自带了通用的日志封装
- spring-jcl替代了通用的日志，仍然支持可重写
- 自动检测log4j 2.x, SLF4J, JUL（java.util.Logging）而不是其他的支持
- 访问Resuouce时提供getFile或和isFile防御式抽象
- 基于NIO的readableChannel也提供了这个新特性

核心容器

- 支持候选组件索引(也可以支持环境变量扫描)
- 支持@Nullable注解
- 函数式风格GenericApplicationContext/AnnotationConfigApplicationContext
- 基本支持bean API注册
- 在接口层面使用CGLIB动态代理的时候，提供事物，缓存，异步注解检测
- XML配置作用域流式
- Spring WebMVC
- 全部的Servlet 3.1 签名支持在Spring-provied Filter实现
- 在Spring MVC Controller方法里支持Servlet4.0 PushBuilder参数
- 多个不可变对象的数据绑定(Kotlin/Lombok/@ConstructorPorties)
- 支持jackson2.9
- 支持JSON绑定API
- 支持protobuf3
- 支持Reactor3.1 Flux和Mono

SpringWebFlux

- 新的spring-webflux模块，一个基于reactive的spring-webmvc，完全的异步非阻塞，旨在使用enent-loop执行模型和传统的线程池模型。
- Reactive说明在spring-core比如编码和解码
- spring-core相关的基础设施，比如Encode 和Decoder可以用来编码和解码数据流；DataBuffer 可以使用java ByteBuffer或者Netty ByteBuf;ReactiveAdapterRegistry可以对相关的库提供传输层支持。

- 在spring-web包里包含HttpMessageReade和HttpMessageWrite

测试方面的改进

- 完成了对JUnit 5’s Juptier编程和拓展模块在Spring TestContext框架
- SpringExtension:是JUnit多个可拓展API的一个实现，提供了对现存Spring TestContext Framework的支持，使用@ExtendWith(SpringExtension.class)注解引用。
- @SpringJunitConfig:一个复合注解
- @ExtendWith(SpringExtension.class) 来源于Junit Jupit
- @ContextConfiguration 来源于Srping TestContext框架
- @DisabledIf 如果提供的该属性值为true的表达或占位符，信号：注解的测试类或测试方法被禁用
- 在Spring TestContext框架中支持并行测试
- 具体细节查看Test 章节 通过SpringRunner在Sring TestContext框架中支持TestNG, Junit5,新的执行之前和之后测试回调。
- 在testexecutionlistener API和testcontextmanager新beforetestexecution()和aftertestexecution()回调。MockHttpServletRequest新增了getContentAsByteArray()和getContentAsString()方法来访问请求体
- 如果字符编码被设置为mock请求，在print()和log()方法中可以打印Spring MVC Test的redirectedUrl()和forwardedUrl()方法支持带变量表达式URL模板。
- XMLUnit 升级到了2.3版本。

35. Spring注解编程模型

介绍

这篇文档是以Spring Framework 4.2作为框架基础编写的，但是，这篇文档是一份还在进行的工作。所以随着时间推移，你会看到这份文档还在更新。

目录

概要

这些年，Spring Framework已经频繁的升级它可以支持的注解、元注解和组合注解。这篇文档旨在帮助开发者（Spring框架使用者、Spring核心框架开发者和Spring全家桶的成员项目开发者）开发和运用Spring注解。

这些是文档目标

这篇文档的主要目的包含以下内容：

- 怎么使用Spring注解。
- 怎么自定义组合注解。
- 怎么查找Spring注解（或者说，理解Spring注解搜索算法的原理）。

这些并非文档目标

这篇文档并不介绍spring经典注解的语义和配置方法。如果想学习经典注解的细节，建议开发者查阅对应的Javadoc或者参考书籍的相应章节。

术语

元注解

元注解是一种标注在别的注解之上的注解。如果一个注解可以标注在别的注解上，那么这个注解已然是元注解。例如，任何需要被文档化的注解，都应该被 `java.lang.annotation` 包中的元注解 `@Documented` 标注。

Stereotype注解

> 译者注：保留Stereotype原生词汇；可理解为模式化注解、角色类注解。

Stereotype注解是一种在应用中，常被用于声明要扮演某种职责或者角色的注解。例如， `@Repository` 注解用于标注任何履行了repository职责角色的类（这种职责角色通常也会被称为Data Access Object或者DAO）。

`@Component` 是被Spring管理的组件的对应注解。任何标注了 `@Component` 的组件都会在spring组件扫描时被扫描到。同样的，任何标注了 被元注解 `@Component` 标注过的注解 的组件，也会在Spring组件扫描时被扫描到。例如， `@Service` 就是一种被元注解 `@Component` 标注过的注解。

Spring核心框架提供了一系列可直接使用的stereotype注解，包括但不限于 `@Component` , `@Service` , `@Repository` , `@Controller` , `@RestController` , and `@Configuration` 。 `@Repository` , `@Service` 等等注解都基于 `@Component` 的更精细化的组件注解。

组合注解

组合注解是一种被一个或者多个元注解标注过的注解，用以撮合多个元注解的特性到新的注解。例如，叫作 `@TransactionalService` 的注解就是被spring的 `@Transactional` 和 `@Service` 共同标注过，而且它把 `@Transactional` 和 `@Service` 的特性结合到了一起。另外，从技术上上讲， `@TransactionalService` 也是一个常见的Stereotype注解。

注解标注形式

一个注解无论是直接标注还是间接标注一个bean，这个注解在java8的 `java.lang.reflect.AnnotatedElement` 类注释中所约定的含义和特性都不会有任何改变。
在Spring中，如果一个元注解标注了其它注解，其它注解标注了一个bean，那么我们就说这个元注解*meta-present*（间接标注了）这个bean。以上文提到的 `@TransactionalService` 为例，我们可以说，`@Transactional` 间接标注了任何一个标注过 `@TransactionalService` 的bean。

成员别名和覆盖

Attribute Alias(成员别名)将注解的一个成员名变为另一个。一个成员有多个别名时，这些别名是平等可交换的。成员别名可以分为以下几种。

- 1.**Explicit Aliases**（明确的别名）：如果一个注解中的两个成员通过 `@AliasFor` 声明后互为别名，那么它们是明确的别名。
- 2.**Implicit Aliases**（隐含别名）：如果一个注解中的两个或者更多成员通过 `@AliasFor` 声明去覆盖同一个元注解的成员值，它们就是隐含别名。
- 3.**Transitive Implicit Aliases**（传递的隐含别名）：如果一个注解中的两个或者更多成员通过 `@AliasFor` 声明去覆盖元注解中的不同成员，但是实际上因为覆盖的传递性导致最终覆盖的是元注解中的同一个成员，那么它们就是可传递的隐含别名。

Attribute Override（成员覆盖）是注解的一个成员覆盖另一个成员。成员覆盖可以分为以下几种。

- 1.**Implicit Overrides**（隐含的覆盖）：注解 `@One` 有成员A，注解 `@Two` 有也有成员A，如果 `@One` 本身是被元注解 `@Two` 标注的，那么按照命名约定，注解 `@One` 中的成员A实际会覆盖注解 `@Two` 中的成员A（用另一种方式说，两个看似不来自不同注解的成员A指向了同一个成员A）。
- 2.**Explicit Overrides**（明确的覆盖）：如果元注解的成员B通过 `@AliasFor` 声明其别名为成员A，那么成员A就是明确的覆盖了成员B。
- 3.**Transitive Explicit Overrides**（传递的明确覆盖）如果注解 `@One` 中的成员A明确覆盖了注解 `@Two` 中的成员B，而且成员B实际覆盖了注解 `@Three` 中的成员C，那么因为覆盖的传递性，所以成员A实际覆盖了成员C。

例子

Spring Composed

> 译者注：`Spring Composed`项目是`spring`的一个社区项目，内含一些有特色的组合注解。可点击下文中的链接下载源码

Spring Composed项目是可以在在spring4.2.1和更高版本中使用的一系列组合注解。你可以在Spring MVC使用像 `@Get` , `@Post` , `@Put` , 和 `@Delet` 这样的注解，也可以在Spring MVC REST应用中使用 `@GetJson` , `@PostJson` 等等注解。

如果你确信已经学习了足够深入的spring-composed例子，汲取了足够多的灵感，然后你可以为spring-Composed项目贡献出由你自定义的组合注解！

声明了 `@AliasFor` 的成员

Spring4.2引入了用以支持声明和查找注解成员别名的第一个版本。 `@AliasFor` 注解可以用来在一个注解中为一对成员互相声明别名，也可以在组合注解中为一个成员声明一个指向另一个元注解成员的别名。

例如，来自spring-test模块的 `@ContextConfiguration` 注解现在是这样定义的：

```
public @interface ContextConfiguration {

    @AliasFor("locations")
    String[] value() default {};

    @AliasFor("value")
    String[] locations() default {};

    // ...
}
```

同样的，组合注解可以使用 `@AliasFor` 来覆盖元注解的成员，这样可以在组合注解中进行精细的操作，而且注解与元注解之间的关系更有层次。实际上，这也提供了为元注解成员起别名的可操作方案。

例如，我们可以使用自己想要的成员名称去开发组合注解，然后再覆盖元注解的成员。如下。

```
@ContextConfiguration
public @interface MyTestConfig {
    @AliasFor(annotation = ContextConfiguration.class, attribute
 = "value")
    String[] xmlFiles();

    // ...
}
```

FAQ

1) `@AliasFor`可以用于标注`@Component`和`@Qualifier`的成员变量吗？

答案是不可以。

在`@Qualifier`和stereotype注解（例如`@Component`，`@Repository`，`@Controller`,和其它定制的stereotype注解）不能被`@AliasFor`影响。原因是这些注解的成员变量在`@AliasFor`被发明之前数年就已经存在了。因此，出于向后的兼容性考虑，这些注解的成员变量就不能使用`@AliasFor`。

附录

使用`@AliasFor`的注解

在Spring4.2版本中，以下注解使用了`@AliasFor`来为它们的成员声明别名。

- `org.springframework.cache.annotation.Cacheable`
- `org.springframework.cache.annotation.CacheEvict`
- `org.springframework.cache.annotation.CachePut`
- `org.springframework.context.annotation.ComponentScan.Filter`
- `org.springframework.context.annotation.ComponentScan` `
- `org.springframework.context.annotation.ImportResource`
- `org.springframework.context.annotation.Scope`

- `org.springframework.context.event.EventListener`
- `org.springframework.jmx.export.annotation.ManagedResource`
- `org.springframework.messaging.handler.annotation.Header`
- `org.springframework.messaging.handler.annotation.Payload`
- `org.springframework.messaging.simp.annotation.SendToUser`
- `org.springframework.test.context.ActiveProfiles`
- `org.springframework.test.context.ContextConfiguration`
- `org.springframework.test.context.jdbc.Sql`
- `org.springframework.test.context.TestExecutionListeners`
- `org.springframework.test.context.TestPropertySource`
- `org.springframework.transaction.annotation.Transactional`
- `org.springframework.transaction.event.TransactionalEventListener`
- `org.springframework.web.bind.annotation.ControllerAdvice`
- `org.springframework.web.bind.annotation.CookieValue`
- `org.springframework.web.bind.annotation.CrossOrigin`
- `org.springframework.web.bind.annotation.MatrixVariable`
- `org.springframework.web.bind.annotation.RequestHeader`
- `org.springframework.web.bind.annotation.RequestMapping`
- `org.springframework.web.bind.annotation.RequestParam`
- `org.springframework.web.bind.annotation.RequestPart`
- `org.springframework.web.bind.annotation.ResponseStatus`
- `org.springframework.web.bind.annotation.SessionAttributes`
- `org.springframework.web.portlet.bind.annotation.ActionMapping`
- `org.springframework.web.portlet.bind.annotation.RenderMapping`

待发掘主题`(译者注：课后习题--!)`_

- 记述 注解和标注了注解和元注解的类、接口、成员方法、成员变量、参数的通用搜索算法。
    - 如果一个注解既是以注解又是以元注解的方式标注了一个元素会发生什么呢？
    - 一个注解标注了 `@Inherited` （包括自定义组合注解）后，是如何对搜索算法产生影响呢？
- 记述 通过 `@AliasFor` 配置注解成员别名的技术原理。

  - 如果一个成员和它的别名都声明在一个注解实例（成员和别名值相同或者不同时）中在技术上会发生什么？
    - 较有代表性的一种情况是，一个 `AnnotationConfigurationException` 将会被抛出。
- 记述组合注解的技术原理。
- 记述 组合注解成员覆盖元注解成员的原理。
  - 详细记述 查找成员的算法原理：
    - 基于命名约定的间接覆盖（换句话说，组合注解中有明确名字和类型的成员去覆盖元注解的成员）
    - 使用@AliasFor来直接覆盖
  - 如果一个成员和它的众多别名中的一个在注解继承的某一层级中被重新声明了会发生什么？哪个会生效？
  - 总之，成员变量声明时的冲突是怎么被解决的？

Spring AOP的经典用法

在本附录中，我们会讨论一些初级的Spring AOP接口，以及在Spring 1.2应用中所使用的AOP支持。 对于新的应用，我们推荐使用 Spring AOP 2.0来支持，在AOP章节有介绍。 但在已有的项目中，或者阅读数据或者文章时，可能会遇到Spring AOP 1.2风格的示例。 Spring 2.0完全兼容Spring 1.2，在本附录中所有的描述都是Spring 2.0所支持的。

Spring中的切点API

一起看一下Spring是如何处理关键切点这个概念。

概念

Spring的切点模型能够让切点重（chong）用不同的独立的增强类型。 这样可以实现，针对不同的增强，使用相同的切点。

`org.springframework.aop.Pointcut` 是一个核心接口，用于将增强定位到特定的类或者方法上。 完整的接口信息如下：

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

将 `Pointcut` 拆分成两个部分，允许重（chong）用类和方法匹配的部分，和细粒度的组合操作（例如和其他的方法匹配器执行一个“组合”操作）。

`ClassFilter` 接口用于将切点限制在给定的目标类上。 如果 `matches()` 方法总是返回true，所有的类都会被匹配上。

```
public interface ClassFilter {

    boolean matches(Class clazz);

}
```

`MethodMatcher` 接口通常更为重要。完整的接口描述如下：

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);

}
```

`matches(Method, Class)` 方法用于测试切点是否匹配目标类的一个指定方法。这个测试可以在AOP代理创建时执行，避免需要在每一个方法调用时，再测试一次。如果对于一个给定的方法， `matches(Method, Class)` 方法返回true，并且对于同MethodMatcher实例的 `isRuntime()` 方法也返回true，那么在每次被匹配的方法执行时，都会调用 `boolean matches(Method m, Class targetClass, Object[] args)` 方法。这样使得在目标增强执行前，一个切点可以在方法执行时立即查看入参。

大部分MethodMatcher是静态的，意味着他们 `isRuntime()` 方法的返回值是false。在这种情况下， `boolean matches(Method m, Class targetClass, Object[] args)` 方法是永远不会被调用的。

提示

> 如果可以，尽量将切点设置为静态，这样在一个AOP代理生成后，可以允许AOP框架缓存评估的结果。

切点操作

Spring在切点的操作：尤其是， `组合（union）` 和 `交叉（intersection）` 。

- 组合意味着方法只需被其中任意切点匹配。
- 交叉意味着方法需要被所有切点匹配。
- 组合通常更为有用。
- 切点可以使用 `org.springframework.aop.support.Pointcuts` 类或者 `org.springframework.aop.support.ComposablePointcut` 中的静态方法组合。 然而，使用AspectJ的切点表达式通常是一种更为简单的方式。

AspectJ切点表达式

自从2.0版以后，Spring所使用的最重要切点类型就是 `org.springframework.aop.aspectj.AspectJExpressionPointcut` 。 这个切点使用了一个AspectJ支持的库，用以解析AspectJ切点表达式的字符串。

有关原始AspectJ切点元素支持的讨论，请参阅之前章节。

方便的切点实现

Spring提供了几个方便的切点具体实现。有些可以在框架外使用；其他的则为应用程序的特定切点实现所需要的子类。

静态切点

静态切点是基于方法和目标类的，不能将方法参数也考虑其中。 对于大多数用法，静态切点是足够且最佳的选择。

对于Spring来说，当一个方法第一次被调用是，对静态切点仅仅评估一次是可行的：在本次评估后，再次调用该方法时，就没有必要再对切点进行评估。

我们一起看一些Spring中包含的静态切点具体实现。

正则表达式切点

一个显而易见的方式是使用正则表达式来指定静态切点。几个在Spring之外的框架可以实现这部分功能。 `org.springframework.aop.support.Perl5RegexpMethodPointcut` 是一个常见的正则表达式切点，使用Perl 5正则表达式语法。 `Perl5RegexpMethodPointcut` 类的正则表达式匹配依赖于Jakarta ORO。 Spring也提供了 `JdkRegexpMethodPointcut` 类，可以在JDK 1.4版本之上使用正则表达式。

使用 `Perl5RegexpMethodPointcut` 类，你可以提供一个正则表达式字符串的列表。 如果与该列表的中的某个正则匹配上了，那么切点的判定就为true（判定的结果是这些切点的有效组合）。

使用方法如下所示：

```
<bean id="settersAndAbsquatulatePointcut"
        class="org.springframework.aop.support.Perl5RegexpMethod
Pointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring提供了一个方便的类， `RegexpMethodPointcutAdvisor` ，允许我们引用一个Advice（记住一个Advice可能是一个介入增强、前置增强、或者异常抛出增强等）。 实际上，Spring会使用 `JdkRegexpMethodPointcut` 类。 使用 `RegexpMethodPointcutAdvisor` 简化配置，这个类封装了切点和增强。如下所示：

```
<bean id="settersAndAbsquatulateAdvisor"
        class="org.springframework.aop.support.RegexpMethodPoint
cutAdvisor">
    <property name="advice">
        <ref bean="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

`RegexpMethodPointcutAdvisor` 可以被任意类型的增强使用。

属性驱动切入

一个重要的静态切点就是 `metadata-driven` 切点。它会使用一些元数据属性信息：通常是源码级的元数据。

动态切点

动态切点的判定代价比静态切点要大。动态切点除了静态信息外，还需要考虑方法 `参数` 。这意味着它们在每次方法调用时都必须进行判定；判定的结果不能被缓存，因为参数是变化的。

代表性的事例是 `控制流` 切点。

控制流切点

Spring的控制流切点在概念上与AspectJ的 `cflow` 切点类似，不过功能稍弱。（目前没有方法，可以指定一个切点在其他切点匹配的连接点后执行。） 一个控制流切点匹配当前的调用栈【待定】。例如，如果一个连接点被一个在 `com.mycompany.web` 包中、或者 `SomeCaller` 类中的方法调用，就会触发。控制流切点使用 `org.springframework.aop.support.ControlFlowPointcut` 类来指定。 说明

> 控制流切点在运行时进行评估明显代价更大，甚至是其他动态切点。在Java 1.4，大概是其他动态切点的5倍。

Pointcut父类

Spring提供了一些有用的切点父类，方便开发者实现自己的切点。

因为静态切点是最为实用的，你可能需要实现StaticMethodMatcherPointcut的子类，如下所示。 这里只需要实现一个抽象方法即可（虽然也可以覆盖其他方法来自定义类的行为）。

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }

}
```

Spring也有动态切点的父类。 在Spring 1.0 RC2版本之后，可以自定义任意增强类型的切点。

自定义切点

由于切点在Spring AOP中都是Java类，而不是语言特征（就像在AspectJ中），可以声明自定义切点，无论静态还是动态。 自定义切点在Spring中是可以任意复杂的。然而，如果可以，推荐使用AspectJ切点表达式。

说明

> Spring之后的版本可能支持由JAC提供的“语义切点”。 例如：在目标对象中，所有修改实例变量的方法。

Spring中的Advice接口

现在让我们看一下Spring AOP如何处理Advice（增强）。

Advice的生命周期

每个Advice都是一个Spring的Bean。一个Advice实例在被增强的对象间共享，或者对于每一个被增强的对象都是唯一的。 这取决于增强是类级的、还是对象级的【待定】。 Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to `per-class` or `per-instance` advice.

Per-class级增强最为常用。它适用于通常的增强，例如事务增强。这种增强不依赖于代理对象或者增加新的状态；它们只是对方法和参数进行增强。 Per-instance级增强适用于介绍，支持它很复杂【待定】。在本示例中，增强对被代理的对象添加了一个状态。

也可以在同一个AOP代理中，使用共享和per-instance级增强的组合。

Spring中的增强类型

Spring在框架层之外，支持多种方式的增强，并且支持任意增强类型的可扩展性。我们一起了解一下标准增强类型和增强的基础概念。

拦截式环绕型增强

Spring中最基本的增强类型之一就是`拦截式环绕型增强`，通过使用方法拦截器，Spring完全符合AOP联盟的环绕型增强接口。环绕型方法拦截器应该实现以下接口：

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;

}
```

`invoke()` 方法的 `MethodInvocation` 表示了将要被调用的方法、目标连接点、AOP代理、以及该方法的参数。 `invoke()` 方法应当返回调用结果：目标连接点的返回值。

一个简单的`方法拦截器`实现如下所示：

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Thr
owable {
        System.out.println("Before: invocation=[" + invocation +
"]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }

}
```

注意调用MethodInvocation对象的 `proceed()` 方法。这个方法将拦截器链路调向连接点。 大多数拦截器会调用该方法，并返回该方法的值。但是，就像任意环绕增强一样，一个方法拦截器也可以返回一个不同的值，或者抛出一个异常，而不是调用 `proceed()` 方法。 但是，没有足够的理由，不要这么干！

说明

> 方法拦截器提供与其他AOP联盟标准的AOP实现的互通性。 在本章剩余部分讨论的其他类型增强，会以Spring特定的方式实现AOP的概念。 使用最为具体的类型增强有一定优势，但如果你想在其他AOP框架中使用切面，就需要坚持使用方法拦截器。 需要注意的是，切点在框架间是不通用的，AOP联盟目前没有定义切点的接口。

前置增强

一个简单的增强类型是前置增强。这种增强不需要一个 `MethodInvocation` 对象，因为它仅仅在方法进入时被调用。

前置增强的优势是不需要调用 `proceed()` 方法，因此不会无故中断调用链。

`MethodBeforeAdvice` 接口如下所示。【待定】 interface is shown below. (Spring's API design would allow for field before advice, although the usual objects apply to field interception and it's unlikely that Spring will ever implement it).

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;

}
```

需要注意的是该方法的返回类型是 `void` 。前置增强可以在连接点钱插入一些自定义的行为，但是不能改变返回结果。 如果一个前置增强抛出一个异常，它会中断调用链中接下来的执行步骤。这个异常将传递到调用链的上一层。 如果该异常没有被处理，或者在被调用方法中签名【待定】，这个异常会直接传递给方法调用方；否则，该异常会被AOP代理类封装到一个未经检查的异常中。

在Spring中，一个前置增强的例子：统计所有方法的执行次数：

```
public class CountingBeforeAdvice implements MethodBeforeAdvice
{

    private int count;

    public void before(Method m, Object[] args, Object target) t
hrows Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

提示

> 前置增强可以被任何切点使用。

异常抛出增强

当连接点返回的结果是一个抛出的异常时，异常抛出增强会被调用。 Spring提供异常抛出增强。 需要主意的是 `org.springframework.aop.ThrowsAdvice` 接口不包括任何方法：它是一个标签式接口，标识给出的对象实现了一个或多个类型的异常抛出增强。 它们的格式如下所示：

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

只有最后一个参数是必须的。这个方法可能拥有1个或者4个参数，取决于增强方法是否对被增强的方法和方法参数感兴趣。 下面的类是异常抛出增强的例子。

如果一个 `RemoteException` （包括子类）被抛出，下面这个增强就会被调用：

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

}
```

如果一个 `ServletException` 被抛出，下面这个增强就会被调用。与上面不同的是，该方法声明了4个参数，因此它可以访问被调用的方法、方法参数和目标对象：

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }

}
```

最后一个示例描述了，一个类中如果声明两个方法，可以同时处理 `RemoteException` 和 `ServletException` 。一个类中可以包含任意个异常抛出增强的处理方法。

```
public static class CombinedThrowsAdvice implements ThrowsAdvice
{

    public void afterThrowing(RemoteException ex) throws Throwab
le {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object ta
rget, ServletException ex) {
        // Do something with all arguments
    }
}
```

说明

> 如果一个异常抛出增强本身抛出了一个异常，它将覆盖掉原始的异常（例如，改变抛给用户的异常）。 这个覆盖的异常通常是一个运行时异常；这样就可以兼容任何的方法签名。 但是，如果一个异常抛出增强抛出了一个检查时异常，这个异常必须和该目标方法的声明匹配，以此在一定程度上与特定的目标签名相结合。

不要抛出与目标方法签名不兼容的检查时异常！

提示

> 异常抛出增强可以被任意切点使用。

后置增强

后置增强必须实现 `org.springframework.aop.AfterReturningAdvice` 接口，如下所示：

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] a
rgs,
            Object target) throws Throwable;

}
```

一个后置增强可以访问被调用方法的返回值（不能修改）、被调用方法、方法参数、目标对象。

下面的后置增强统计了所有执行成功的方法调用，即没有抛出异常的调用：

```
public class CountingAfterReturningAdvice implements AfterReturn
ingAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Obj
ect[] args,
                    Object target) throws Throwable {
                ++count;
            }

    public int getCount() {
        return count;
    }

}
```

这个增强不会改变执行路径。如果它抛出了一个异常，该异常会抛出到拦截链，而不是返回返回值。

提示

> 后置增强可以被任意切点使用。

引介增强

Spring将引介增强当作一个特殊的拦截式增强。

引介增强需要一个 `IntroductionAdvisor` 和一个 `IntroductionInterceptor` 实现以下接口：

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);

}
```

`invoke()` 方法继承自AOP联盟的 `MethodInterceptor` 接口，必须被引介实现：也就是说，如果被调用的方式是一个被介入的接口，该引介拦截器就会负责处理该方法的调用，不能调用 `proceed()` 方法。

不是所有的切点都可以使用引介增强，因为它只适用于类级，而不是方法级。你可以通过 `IntroductionAdvisor` 来使用引介增强，该类有如下几个方法：

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;

}

public interface IntroductionInfo {

    Class[] getInterfaces();

}
```

没有 `MethodMatcher`，因此也没有 `Pointcut` 与引介增强相关联。只有类过滤器是符合逻辑的。 `getInterfaces()` 方法会返回被该增强器引介的接口集合。`validateInterfaces()` 会在内部被调用，用于确定被引介的接口是否可以被配置的 `IntroductionInterceptor` 所实现。

让我们一起看一个Spring测试套件的简单示例。假设我们想要将以下的接口介入到一个或多个对象中：

```
public interface Lockable {

    void lock();

    void unlock();

    boolean locked();

}
```

这里解释了一个 `mixin` 。我们希望能够将被增强的对象转换成一个Lockable对象，无论它原来的类型是什么，并且调用转换后对象的lock和unlock方法。如果调用lock()方法，我们希望所有的setter方法抛出一个 `LockedException` 异常。这样我们就可以提供一个切面，使该对象不可变，而不需要对该对象有所了解：一个很好的AOP示例。

首先，我们需要一个 `IntroductionInterceptor` ，这很重要。在这种情况下，我扩展 `org.springframework.aop.support.DelegatingIntroductionInterceptor` 类。我们可以直接实现IntroductionInterceptor，但是大多数情况下使用 `DelegatingIntroductionInterceptor` 是最合适的。

`DelegatingIntroductionInterceptor` 被设计成代理一个需要被引介接口的真实实现，隐藏使用拦截器去这样做。使用构造函数的参数，可以把代理设置为任意对象；默认的代理（使用无参构造函数时）就是引介增强【待定】。The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. 因此在下面的示例中，代理是 `DelegatingIntroductionInterceptor` 的子类 `LockMixin` 。给定的代理（默认是自身），一个 `DelegatingIntroductionInterceptor` 对象查找所有被该代理所实现的接口结合（除了IntroductionInterceptor），并支持代理介入它们。像 `LockMixin` 的子类调用 `suppressInterface(Class intf)` 方法，可以禁止不能被暴露的接口被调用。然而无论一个 `IntroductionInterceptor` 准备支持多少个接口， `IntroductionAdvisor` 都会控制哪些接口实际是被暴露的。一个被引介的接口会隐藏掉目标对象的所有接口的实现。

因此 `DelegatingIntroductionInterceptor` 的子类LockMixin，也实现了Lockable接口本身。 超类会自动获取Lockable能支持的引介，因此我们不需要为此设置。这样我们就可以引介任意数量的接口。

需要注意所使用的 `locked` 对象变量。它有效的增加了目标对象的附加状态。

```
public class LockMixin extends DelegatingIntroductionInterceptor
implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Thr
owable {
        if (locked() && invocation.getMethod().getName().indexOf(
"set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }

}
```

通常是不需要覆盖 `invoke()` 方法的：如果方法被引介的话， `DelegatingIntroductionInterceptor` 代理会调用方法，否则调用连接点，通常也是足够了。 在这种情况下，我们需要加入一个检查：如果处于锁住的模式，任何setter方法都是不能被调用。

所需要的引介增强器非常简单。它所需要做的仅仅是持有一个明确的 `LockMixin` 对象，指定需要被引介的接口（在本示例中，仅仅是 `Lockable` 接口）。一个更加复杂的例子是持有一个引介拦截器的引用（被定义为一个原型）：在本示例中，没有配置和一个 `LockMixin` 对象相关，所有我们简单使用 `new` 来创建。

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor
{

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }

}
```

我们可以非常简单的使用这个增强器：不需要任何配置。（但是，使用 `IntroductionInterceptor` 的同时，不使用 `IntroductionAdvisor` 是不行的。）和之前介绍的一样，Advisor是一个per-instance级的，它是有状态的。因此对于每一个被增强的对象，就像 `LockMixin` 一样，我们都需要一个不同的 `LockMixinAdvisor` 。Advisor就是被增强对象状态的一部分。

我们可以使用编程的方式应用这个Advisor，使用 `Advised.addAdvisor()` 方法，或者在XML中配置（推荐），就像其他Advisor一样。下面会讨论所有代理创建的选择方式，包括“自动代理创建者”正确的处理引介和有状态的mixins。

Spring中的Advisor接口

在Spring中，一个Advisor是一个切面，仅仅包括了一个和切点表达式相关联的增强对象。

除了介绍的特殊情况，任何Advisor都可以被任意增强使用。`org.springframework.aop.support.DefaultPointcutAdvisor` 是最为常用的advisor类。例如，它可以被 `MethodInterceptor` 、 `BeforeAdvice` 和 `ThrowsAdvice` 使用。

在Spring的同一个AOP代理中，有可能会混淆Advisor和增强。 例如，在一个代理的配置中，你可能使用了一个拦截式环绕增强、异常抛出增强和前置增强：Spring会自动创建需要的拦截链。

使用ProxyFactoryBean创建AOP代理

如果你的业务对象使用了Spring IoC容器（一个ApplicationContext或者BeanFactory），你应该、也会希望使用一个Spring的AOP FactoryBean。 （需要注意的是，一个FactoryBean间接的引入了一层，该层可以创建不同类型的对象。）

说明

> Spring 2.0 AOP在内部也是用了工厂对象。

在Spring中，创建AOP代理最基础的方式是使用 `org.springframework.aop.framework.ProxyFactoryBean` 类。 这样可以完全控制将要使用的切点和增强，以及它们的顺序。 然而，更简单的是这是可选的，如果你不需要这样的控制。

基础

`ProxyFactoryBean` 就像Spring其他 `FactoryBean` 的实现一样，间接的引入了一个层次。 如果你定义了一个名为 `foo` 的 `ProxyFactoryBean` ，那么对象引用的 `foo` ，不是 `ProxyFactoryBean` 实例本身， 而是 `ProxyFactoryBean` 对象调用 `getObject()` 方法的返回值。 这个方法会创建一个AOP代理来包装目标对象。

使用一个 `ProxyFactoryBean` 或者IoC感知类来创建AOP代理的最大好处之一是，增强和切点同样也可以被IoC管理。 这是一个强大的功能，实现的方法是其他AOP框架难以企及的。

JavaBean属性

与大多数Spring所提供的 `FactoryBean` 实现相同的是， `ProxyFactoryBean` 本身也是一个JavaBean。 它的属性用于：

- 指定你想要代理的目标
- 指定是否需要使用CGLIB（参照下面的介绍和基于JDK和CGLIB的代理,）

一些关键的属性继承自 `org.springframework.aop.framework.ProxyConfig` （Spring中所有代理工厂的超类） 这些关键属性包括：

- `proxyTargetClass`：如果目标类将被代理标志为 `true` ，而不是目标类的接口。 如果该属性设置为 `true` ，CGLIB代理就会被创建（但也需要参见基于JDK和CGLIB的代理）
- `optimize`：控制是否积极优化通过 `CGLIB` 创建的代理类。除非完全了解AOP代理相关的优化处理，否则不要使用这个设置。 这个设置当前只对CGLIB代理有效；对JDK动态代理无效。
- `frozen`：如果一个代理配置是 `frozen` ，那么就不再允许对该配置进行更改。 如果你不想调用者在代理创建后操作该代理（通过被增强的接口），作为轻微的优化手段是该配置是很有用的。 该配置的默认值是 `false` ，因此增加附带的advice是允许的。
- `exposeProxy`：该属性决定当前的代理是否在暴露在 `ThreadLocal` 中，让它可以被目标对象访问到。 如果一个目标对象需要获取该代理， `exposeProxy` 就设置为 `true` ，目标对象可以通过 `AopContext.currentProxy()` 方法获取当然的代理。
- `aopProxyFactory`：需要使用的 `AopProxyFactory` 实现。提供了是否使用动态代理的自定义方式，CGLIB或者其他代理模式。 该属性的默认是适当的选择动态代理或者CGLIB。没有必要使用该属性；在Spring 1.1中它的目的在于添加新的代理类型。

`ProxyFactoryBean` 的其他属性：

- `proxyInterfaces`：接口名称的字符串数组。如果没有提供该属性，会使用一个CGLIB代理参见（基于JDK和CGLIB的代理）

- `interceptorNames`： `Advisor` 字符串数组，需要使用的拦截器或者其他advice的名称。 顺序非常重要，先到的先处理。也就是列表中的第一个拦截器将会第一个处理调用。

这些名称是当前工厂的实例名称，包括从祖先工厂继承来的名称。 这里不能包括bean的引用，因为这么做的结果是 `ProxyFactoryBean` 忽略advice的单例设置。

你可以在一个拦截器名称后添加一个星号(`*`)。这样在应用中，所有以型号前的部分为名称开始的advisor对象，都将被应用。 这个特性的示例可以在使用'全局'advisor中找到。

- singleton: 是否该工厂返回一个单例对象，无论调用多少次 `getObject()` 方法。 某些 `FactoryBean` 实现提供了这样的方法。该配置的默认值是 `true` 。 如果你需要使用一个有状态的advice，例如有状态的mixins，使用prototype的advice，以及将该属性设置为 `false` 。

基于JDK和CGLIB的代理

本章作为明确的文档，介绍 `ProxyFactoryBean` 对于一个特定的目标对象（即被代理的对象）如何选择创建一个基于JDK的还是基于CGLIB的代理。

说明

> `ProxyFactoryBean` 创建基于JDK或基于CGLIB的代理在Spring 1.2.x和2.0版本间有所改变。 `ProxyFactoryBean` 目前与 `TransactionProxyFactoryBean` 类的自动检测接口所表现的语义相似。

如果被代理的目标对象的类（以下简称目标类）没有实现任何接口，那么就会创建基于CGLIB的代理。 这是一个最简单的情景，因为JDK代理是基于接口的，没有接口就意味着JDK代理类是行不通的。 即一个简单的目标类插入，通过 `interceptorNames` 属性指定一系列的拦截器。 需要注意的是即使 `ProxyFactoryBean` 的 `proxyTargetClass` 属性被设置为 `false` ，也会创建基于CGLIB的代理。 （这显然没有任何意义，而且最好从Bean定义中移除，因为它是冗余的，而且是很糟的混淆。）

如果目标类实现了一个（或者多个）接口，那么被创建代理的类型取决于 `ProxyFactoryBean` 的配置。 如果 `ProxyFactoryBean` 的 `proxyTargetClass` 属性被置为 `true` ，那么会创建基于CGLIB的代理。 这很有道理，并且符合最小惊讶原则。 即使 `ProxyFactoryBean` 的 `proxyInterfaces` 属性被设置成一个或多个全量的接口名称，只要 `proxyTargetClass` 属性被置为 `true` ，就会创建基于CGLIB的代理。

即使 `ProxyFactoryBean` 的 `proxyInterfaces` 属性被设置成一个或多个全量的接口名称，那么就会创建基于JDK的代理。 被创建的代理会实现所有 `proxyInterfaces` 所指定的接口；如果目标类也实现的接口多余 `proxyInterfaces` 所指定的，这也是可以的，但这些额外的接口不会被创建的代理所实现。

如果 `ProxyFactoryBean` 的 `proxyInterfaces` 没有被设置，但是目标类也没有实现一个（或多个）接口， `ProxyFactoryBean` 会自动检测至少一个目标类实际实现的接口，并且创建一个基于JDK的代理。 实际上被代理的接口，就是目标类所有实现的接口；事实上，这和简单的将目标类实现的每一个接口所组成的列表设置为 `proxyInterfaces` 属性，效果是一样的。 然而，自动检测显然减少了工作量，也不容易出现拼写错误。

代理接口

我们一起看一个简单的 `ProxyFactoryBean` 示例。这个例子涉及：

- 一个将被代理的目标对象。在下面的示例中定义的是"personTarget"对象。
- 一个Advisor和一个Interceptor用以提供增强。
- 一个AOP代理对象指定了目标对象（"personTarget"对象）和需要代理的接口，以及应用的advice。

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property
value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.inter
ceptor.DebugInterceptor">
</bean>

<bean id="person" class="org.springframework.aop.framework.Proxy
FactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person
</value></property>
    <property name="target"><ref bean="personTarget"/></property>

    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

需要主意的是 `interceptorNames` 属性使用的是一个字符串列表：当前工厂的interceptor或者advisor名称。 Advisor、拦截器、前置增强、后置增强、异常抛出增强都可以被使用。Advisor的排序很重要。

说明

> 你可能会疑惑，为什么列表没有持有bean的引用。 原因是如果一个ProxyFactoryBean的singleton属性是false，它就必须返回一个独立的代理对象。 如果每一个advisor对象本身是一个prototype的，就应该返回一个独立的对象，因此从工厂中获得一个prototype的实例是有必要的；持有一个引用是不行的。

上面定义的"person"对象可以被一个Person实现所替代，如下所示：

```
Person person = (Person) factory.getBean("person");
```

在同一个IoC上下文中的其他bean，也可以强类型依赖它，作为一个原生的java对象：

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person" /></property>
</bean>
```

本示例中的 `PersonUser` 类暴露了一个Person类型的属性。 就此而言，AOP代理可以透明的替代一个“眞实”person的实现。 然而，它的class是一个动态代理类。它也可以被强制转换为 `Advised` 接口（接下来会讨论）。

可以使用内部匿名bean来隐藏目标和代理的区别。 只有 `ProxyFactoryBean` 的定义是不一样的；包含的advice仅仅是为了示例的完整性：

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property
value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.inter
ceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.Proxy
FactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person
</value></property>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name"><value>Tony</value></property>
            <property name="age"><value>51</value></property>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

这样做有一个好处是只会有一个 `Person` 类型的对象：如果我们想要阻止用户从应用上下文中获取一个没有被advise的对象是很有用的， 或者需要阻止Spring IoC容器的自动注入时的歧义。 还有一个可以作为优点的是，ProxyFactoryBean定义是独立的。 但是，有时候从工厂中可以得到一个没有被advise的目标也是一个优点：比如在特定的测试场景。

代理类

如果你需要代理一个类，而不是代理一个或多个接口？

设想一下，在上面的实例中，如果没有 `Person` 接口，我们需要去增强一个叫 `Person` 的类，该类没有实现任何业务接口。 在这种情况下，你需要配置Spring，使用CGLIB代理，而不是动态代理。 只需要将ProxyFactoryBean的 `proxyTargetClass` 属性置为true。 虽然最好使用接口变成，而不是类，但当增强遗留的代码时，增强目标类而不是目标接口，可能更为有用。 （通常情况下，Spring不是约定俗成的。它对应用好的实践非常简单，并且避免强制使用特定的实践方式）

如果需要，你可以在任何情况下强制使用CGLIB，甚至对于接口。

CGLIB代理的的工作原理是在运行时生成目标类的子类。 Sprig将原始目标对象的方法调用委托给该生成的子类：该子类使用了装饰器模式，在增强时织入。

CGLIB代理通常对用户是透明的。然而，有一些问题需要考虑：

- `Final` 方法是不能被advise的，因为它们不能被重写。
- 从Spring 3.2之后，就不再需要在项目的classpath中加入CGLIB的库，CGLIB相关的类已经被重新打包在org.springframework包下，直接包含在prig-core 的jar包中。 这样即方便使用，又不会和其他项目所依赖的CGLIB出现版本冲突。

CGLIB代理和动态代理在性能上有所差异。 自从Spring 1.0后，动态代理稍快一些。 然而，这种差异在未来可能有所改变。 在这种情况下，性能不再是考虑的关键因素。

使用全局advisor

通过在拦截器的名称上添加星号，所有匹配星号前部分的名称的advisor，都将添加到advisor链路中。 如果你需要添加一个套标准的全局advisor，这可能会派上用场。0

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyF
actoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.intercept
or.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.int
erceptor.PerformanceMonitorInterceptor"/>
```

简明的代理定义

特别是在定义事务代理时，最终可能有许多类似的代理定义。使用父、子bean定义，以及内部bean定义，可能会使代理的定义更加清晰和简明。

首先，创建一个代理的父模板定义。

```
<bean id="txProxyTemplate" abstract="true"
        class="org.springframework.transaction.interceptor.Trans
actionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"
/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

这个定义自身永远不会实例化，所以实际上是不完整的定义。然后每个需要被创建的代理，只需要一个子bean的定义，将目标对象包装成一个内部类定义，因为目标对象永远不会直接被使用。

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MyServiceImpl">
        </bean>
    </property>
</bean>
```

当然也可以覆盖父模板的属性，例如在本示例中，事务传播的设置：

```
<bean id="mySpecialService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MySpecialServic
eImpl">
        </bean>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>

            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop
>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop
>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

需要主意的是，在上面的示例中，我们通过 `abstract` 属性明确的将父bean标记为抽象定义， 就如前面介绍的子bean定义，因此该父bean永远不会被实例化。 应用上下文（不是简单的bean工厂）默认会预先实例化所有单例。 因此，重要的是，如果你有一个仅仅想作为模板的bean（父bean）定义，并且指定了该bean的class， 那么你必须保证该bean的 `abstract` 属性被置为 `tue` ，否则应用上下文会尝试在实际中预先实例化该bean。

使用ProxyFactory以编程的方式创建AOP代理

使用Spring以编程的方式创建AOP代理非常简单。 这也运行你在不依赖Spring IoC容器的情况下使用Spring的AOP。

下面的代码展示了使用一个拦截器和一个advisor创建一个目标对象的代理。

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl)
;
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy(
);
```

第一步是构造一个 `org.springframework.aop.framework.ProxyFactory` 对象。 像上面的示例一样，可以使用一个目标对象创建它，或者使用指定接口集的构造函数替代来创建该ProxyFactory。

你可以添加拦截器和advisor，并在ProxyFactory的生命周期中操作它们。 如果你添加一个IntroductionInterceptionAroundAdvisor，可以使得该代理实现附加的接口集合。

在ProxyFactory也有一些好用的方法（继承自 `AdvisedSupport` ），允许你天机其他的增强类型，比如前置增强和异常抛出增强。 AdvisedSupport是ProxyFactory和ProxyFactoryBean的超类。

提示

> 在IoC框架中集成AOP代理的创建在大多数应用中是最佳实践。 通常，我们推荐在Java代码之外配置AOP。

操作被增强的对象

当你创建了AOP代理，你就能使用 `org.springframework.aop.framework.Advised` 接口来操作他们。 任何一个AOP代理，都能强制转换成该接口，或者无论任何该代理实现的接口。 这个接口包含以下方法：

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException
;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigExcept
ion;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException
;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigExc
eption;

boolean isFrozen();
```

`getAdvisors()` 方法会返回添加到该工厂的每一个advisor、拦截器或者其它类型的增强。如果你添加了一个Advisor，那么返回Advisor数组在该索引下的对象，就是你添加的那个。如果你添加的是一个拦截器或者其他类型的增强，Spring将会把它包装成一个带有切点（切点判断恒为真）的Advisor。如果你添加了 `MethodInterceptor` 对象，该advisor `getAdvisors()` 方法返回值，该索引处会是一个 `DefaultPointcutAdvisor` 对象，该对象包括了你添加的 `MethodInterceptor` 对象和一个匹配所有类和方法的切点。

`addAdvisor()` 可以用于添加任何Advisor。通常该advisor是一个普通的 `DefaultPointcutAdvisor` 对象，包括了切点和advice，可以和任何advice或切点一起使用（除了引介增强）。

默认情况下，在一个代理被创建后，也可以添加或者删除advisor和拦截器。 唯一的限制是，不能增加或者删除一个引介advisor，因为已经从工厂生成的代理不能再进行接口修改。（你可以从工厂中重新获取一个新的代理来避免该问题）。

一个简单的例子是强制转换一个AOP代理成为 `Advised` 对象，并且检验和操作它的advice：

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws a
dvice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut,
 myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.
getAdvisors().length);
```

说明

> 问题是，是否建议（没有一语双关）在生产环境中对一个业务对象进行修改advice，尽管这毫无疑问是一个合法的使用案例。 然而，在开发环境是非常有用的：例如，在测试过程中。 我有时发现将一个拦截器或者advice增加到测试代码中是非常有用的，进入到一个方法中，调用我想测试的部分。（例如，advice可以进入到一个方法的事务中：例如运行一个SQL后检查数据库是否正确更新，在该事务标记回滚之前。）

根据你创建的代理，通常你可以设置一个 `frozen` 标志，在这种情况下， `Advised` 的 `isFrozen()` 方法会返回true， 并且任何通过添加或者删除方法试图修改advice都会抛出一个 `AopConfigException` 异常。 在一些情况下，冻结一个advise对象的状态是有用的，例如，阻止调用代码删除安全拦截器。 在Spring 1.1也用于积极优化，当运行时的修改被认为是没必要的。

使用“autoproxy”能力

至此我们已经考虑过使用一个 `ProxyFactoryBean` 或者相似的工厂类创建明确的AOP代理。

Spring允许我们使用“autoproxy”bean定义，可以自动代理选择的bean定义。 这是建立在Spring“bean后处理器（BeanPostProcessor）”机制之上，这可以允许在容器加在后修改任何bean定义。

在这个模型上，你可以在bean定义的XML文件中设置一些特殊的bean定义，用以配置自动代理机制。 这允许你只需要声明符合代理条件的目标即可：你不需要使用 `ProxyFactoryBean` 。

有两种方式实现：

- 在当前上下文中，使用一个指定了目标bean定义的自动代理创建器，
- 一些特殊自动代理创建器需要分开考虑；由源码级元数据信息驱动的自动代理创建器。

自动代理Bean的定义

`org.springframework.aop.framework.autoproxy` 包提供了以下标准自动代理创建器

BeanNameAutoProxyCreator

`BeanNameAutoProxyCreator` 类是一个 `BeanPostProcessor` ，为纯文本或者通配符匹配出的命名为目标bean自动创建AOP代理。

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames"><value>jdk*,onlyJdk</value></property>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

和 `ProxyFactoryBean` 一样，有一个 `interceptorNames` 属性，而不是一个列表拦截器， 确保原型advisor正确的访问方式。 命名为 "interceptors",可以使任何advisor或者任何类型的advice。

和通常的自动代理一样，使用 `BeanNameAutoProxyCreator` 的要领是，使用最小的配置量，将相同的配置一致地应用到多个对象上。

与bean定义匹配的命名，比如上面示例中的"jdkMyBean"和"onlyJdk"，就是目标类普通的原有bean定义。 一个AOP代理会被 `BeanNameAutoProxyCreator` 自动创建。相同的advice会被应用到所有匹配的bean上。 需要注意的是，被应用的advisor（不是上面示例中的拦截器），对不同的bean可能使用不同的切点。

DefaultAdvisorAutoProxyCreator

一个更一般且更强大的自动代理创建器是 `DefaultAdvisorAutoProxyCreator` 。在上下文中会自动应用符合条件的advisor，不需要在自动代理创建器的bean定义中指定目标对象的bean名称。 它也提供了相同的有点，一致的配置和避免重复定义 `BeanNameAutoProxyCreator` 。

使用此机制涉及：

- 指定一个 `DefaultAdvisorAutoProxyCreator` bean定义。
- 在相同或者相关的上下文中指定一系列的Advisor。需要注意的是，这些都必须是Advisor，而不仅仅是拦截器或者其他的advice。 这很必要，因为这里必须有评估的切点，以便检测候选bean是否符合每一个advice。

`DefaultAdvisorAutoProxyCreator` 会自动的评估包含在每一个advisor中的切点，用以确定每一个业务对象需要应用的advice（就像示例中的"businessObject1"和"businessObject2"）。

这意味着任意数量的advisor都能自动的应用到每一个业务对象。 如果所有在advisor的切点都不能匹配一个业务对象中的任何方法，这个对象就不会被代理。 由于bean的定义都是添加给创建的业务对象。如果需要，它们都会被自动代理。

通常情况下，自动代理都有使调用方或者依赖方不能获取未被advise对象的优点。

在本ApplicationContext中调用getBean("businessObject1")会返回一个AOP代理，而不是目标业务对象（之前的“内部bean”也提供了这种优点）。

```
<bean class="org.springframework.aop.framework.autoproxy.Default
AdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.Transac
tionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInte
rceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">

    <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"
/>
```

如果你想对许多业务对象应用相同的advice， `DefaultAdvisorAutoProxyCreator` 将会有所帮助。 一旦基础定义设置完成，你就可以简单添加新的业务对象，不需要特定的proxy配置。 你也可以轻松地添加其他切面。例如，使用最小的配置修改，添加跟踪或性能监控切面。

DefaultAdvisorAutoProxyCreator提供过滤（使用命名约定，以便只有特定的advisor被评估，允许在相同的工厂中使用多个、不同的被配置的AdvisorAutoProxyCreator）和排序的支持。 Advisor可以实现 `org.springframework.core.Ordered` 接口，当顺序是一个问题时，确保正确的顺序。 在上面示例中使用的TransactionAttributeSourceAdvisor，有一个可配置的顺序值；默认配置是无序的。

AbstractAdvisorAutoProxyCreator

AbstractAdvisorAutoProxyCreator是DefaultAdvisorAutoProxyCreator的超类。 你可以通过继承这个类创建自己的自动代理创建器， 虽然这种情况微乎其微，advisor定义为 `DefaultAdvisorAutoProxyCreator` 框架的行为提供了有限的定制。

使用元数据驱动

一个特别重要的自动代理类型就是元数据驱动。这和 .NET 的 `ServicedComponents` 编程模型类似。 事务管理和其他企业服务的配置在源码属性中保存，而不是像在EJB中一样使用XML部署描述符。

在这种情况下，你结合能够解读元数据属性的Advisor，使用 `DefaultAdvisorAutoProxyCreator` 。 元数据细节存放在备选advisor的切点部分，而不是自动创建器类的本身中。

这实际上是 `DefaultAdvisorAutoProxyCreator` 的一种特殊情况，但值得考虑（元数据感知代码在advisor切点中，而不是AOP框架自身上）。

JPetStore示例应用程序的 `/attributes` 目录，展示了属性驱动的使用方法。 在这种情况下，没必要使用 `TransactionProxyFactoryBean` 。 简单在业务对象上定义事务属性就足够了，因为使用的是元数据感知切点。 包含了下面代码的bean定义，在 `/WEB-INF/declarativeServices.xml` 文件中。 需要注意的是这是通用的，也可以在JPetStore之外使用。

```
<bean class="org.springframework.aop.framework.autoproxy.Default
AdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.Transac
tionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInte
rceptor"/>
</bean>

<bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.Trans
actionInterceptor">
    <property name="transactionManager" ref="transactionManager"
/>
    <property name="transactionAttributeSource">
        <bean class="org.springframework.transaction.interceptor
.AttributesTransactionAttributeSource">
            <property name="attributes" ref="attributes"/>
        </bean>
    </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.common
s.CommonsAttributes"/>
```

`DefaultAdvisorAutoProxyCreator` bean（命名不是重点，甚至可以省略）定义会获取所有在当前应用上下文中符合的切点。在这种情况下， `TransactionAttributeSourceAdvisor` 类星的"transactionAdvisor" bean定义，将适用于携带了事务属性的类或者方法。TransactionAttributeSourceAdvisor通过构造函数依赖一个TransactionInterceptor对象。本示例中通过自动装配解决该问题。 `AttributesTransactionAttributeSource` 依赖一个 `org.springframework.metadata.Attributes` 接口的实现。在本代码片段中，"attributes" bean满足这一点，使用Jakart aCommons Attributes API来获取属性信息。（这个应用代码必须使用Commons Attributes编译任务编译）

JPetStore示例应用程序的 `/annotation` 目录包含了一个类似自动代理的示例，需要JDK 1.5版本以上的注解支持。下面的配置可以自动检测Spring的 `Transactional` ，为包含该注解的bean配置一个隐含的代理。

```
<bean class="org.springframework.aop.framework.autoproxy.Default
AdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.Transac
tionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInte
rceptor"/>
</bean>

<bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.Trans
actionInterceptor">
    <property name="transactionManager" ref="transactionManager"
/>
    <property name="transactionAttributeSource">
        <bean class="org.springframework.transaction.annotation.
AnnotationTransactionAttributeSource"/>
    </property>
</bean>
```

这里定义的 `TransactionInterceptor` 依赖一个 `PlatformTransactionManager` 定义，没有被包含在这个通用文件中（尽管可以包含），因为它对应用的事务需求是定制的（通常是JTA，就像本示例，或者是Hibernate、JDBC）：

```
<bean id="transactionManager"
        class="org.springframework.transaction.jta.JtaTransactio
nManager"/>
```

提示

> 如果你只需要声明式事务管理，使用这些通用的XML定义会导致Spring为所有包含事务属性的类或方法创建自动代理。你不需要直接使用AOP，以及.NET和ServicedComponents相似的编程模型。

这种机制是可扩展的，可以基于通用属性自动代理。你需要：

- 定义你自己的个性化属性。

- 指定包含必要advice的Advisor，包括一个切点，该切点会被一个类或方法上存在的定义属性所触发。 你也可以使用一个已有的advice，仅仅实现了获取自定义属性的一个静态切点。

对每个被advise的类，这样的advisor都可能是唯一的（例如mixins【待定】）：它们的bean需要被定义为prototype，而不是单例。 例如，Spring测试套件中的 `LockMixin` 引介拦截器，可以对一个mixin目标与一个属性驱动切点一起使用。 我们使用通用的 `DefaultPointcutAdvisor` ，使用JavaBean属性进行配置。

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
        scope="prototype"/>

<bean id="lockableAdvisor" class="org.springframework.aop.suppor
t.DefaultPointcutAdvisor"
        scope="prototype">
    <property name="pointcut" ref="myAttributeAwarePointcut"/>
    <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

如果该属性感知切点匹配了 `anyBean` 或者其他bean定义的任何方法，这个mixin都会被应用。 需要注意的是 `lockMixin` 和 `lockableAdvisor` 都是prototype的。 `myAttributeAwarePointcut` 切点可以是一个单例定义，因为它不会持有被advise对象的个性状态。

使用TargetSources

Spring提供了一个 `TargetSource` 概念，由 `org.springframework.aop.TargetSource` 接口所表示。 该接口负责返回实现了连接点的目标对象。 【待定】每次AOP代理处理一个方法调用时，TargetSource`实现都需要一个目标的实例。

开发人员使用Spring AOP通常不需要直接使用TargetSource，但是它提供了一个强大的供给池、热替换和其他复杂的目标。 例如，一个池化的TargetSource可以为每次调用返回不同的目标示例，通过池子来管理这些实例。

如果你没有指定一个TargetSource，默认的实现手段是使用一个包装的本地对象。每次调用返回的是同一个目标（如你所愿）。

让我们看一个Spring提供的标准TargetSource，以及如何使用它们。

提示

> 当使用一个自定义的TargetSource时，你的目标通常是一个prototype bean定义，而不是单例bean定义。 这允许Spring在需要时创建一个新的目标实例。

热替换TargetSource

`org.springframework.aop.target.HotSwappableTargetSource` 的存在，允许一个AOP代理的目标进行切换，同时允许调用者持有她们的引用。

修改TargetSource的目标对象会立即生效。 `HotSwappableTargetSource` 是线程安全的。

你可以如下所示，通过 `HotSwappableTargetSource` 的 `swap()` 方法修改目标对象：

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

需要参考的XML定义如下所示：

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="swapper"/>
</bean>
```

上面的调用的 `swap()` 方法，修改了 `swappable` 对象的目标对象。持有这个bean引用的客户端对此更改毫无感知，但是会立即开始命中新的目标对象。

尽管这个示例没有添加任何advice，并且添加一个advice到一个使用的 `TargetSource` 中也是没必要的，当然任何 `TargetSource` 都可以和任意的advice结合使用。

池化TargetSources

使用一个池化的 `TargetSource` ，提供了一个与无状态会话的EJB类似的编程模型，池子中维护了相同类型的实例，当方法调用时释放池子中的对象。

Spring池子和SLSB池子的关键区别在于，Spring池子可以适用于任意POJO类。通常和Spring一样，这个服务可以用非侵入性的方式使用。

Spring提供了对框架外Commons Pool 2.2的支持，Commons Pool 2.2提供了一个相当有效的池子实现。使用该特性，需要在应用的classpath中加入commons-pool的jar包。也可以继承 `org.springframework.aop.target.AbstractPoolingTargetSource` ，来支持任意其他的池子API。

说明

> Commons Pool 1.5版本以上也被支持，不过在Spring Framework 4.2被弃用了。

示例配置如下所示：

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject" scope="prototype">
    ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPool2TargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
    <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

需要主意的是目标对象，即本示例中的"businessObjectTarget"必须是prototype的。这允许 `PoolingTargetSource` 在需要的时候创建为目标对象创建新的实例来扩张池子大小。参考 `AbstractPoolingTargetSource` 的Javadoc文档，以及你要使用的具体的子类属性信息："maxSize"是最基础的，需要保证它存在

在本示例中，"myInterceptor"是一个拦截器的名称，需要在同一个IoC上下文中被定义。然而，不需要为使用的池子，指定拦截器。如果你只需要池子，不需要任何advice，就不要设置interceptorNames属性。

也可以通过Spring配置将任意池化的对象强制转成 `org.springframework.aop.target.PoolingConfig` 接口，通过一个引介，可以显示当前池子的配置和大小信息。你需要定义一个像这样的advisor：

```
<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>

</bean>
```

这个advisor通过调用 `AbstractPoolingTargetSource` 类中的一个方法方法获取，因此使用MethodInvokingFactoryBean。 这个advisor的命名（本示例中的"poolConfigAdvisor"）必须在ProxyFactoryBean暴露的池化对象的拦截器列表中。

强制转化如下所示：

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

说明

> 池化无状态的服务实例通常是不必要的。 我们不认为这是默认选择，因为大多数的无状态对象自然是县城安全的，并且如果资源被缓存，实例池会存在问题。

简单的池子也可以使用自动代理。可以使用任何自动代理创建器设置TargetSource。

Prototype类型的TargetSource

设置一个"prototype"的TargetSource和池化一个TargetSource是类似的。 在本示例中，当每个方法调用时，都会创建一个目标的示例。 尽管在现代JVM中创建一个对象的成本不高，绑定一个新对象（满足IoC的依赖）可能花费更多。 因此，没有一个很好的理由，你不应该使用这个方法，。

为此，你可以修改上面定义的 `poolTargetSource` 成如下所示（为了清楚起见，我也修改了命名）：

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

只有一个属性：目标bean的命名。在TargetSource实现中使用继承是为了确保命名的一致性。 与池化TargetSource一样，目标bean的定义也必须是prototype。

ThreadLocal的TargetSource

如果你需要为每一个进来的请求（每个线程一个那种）创建一个对象，那么 `ThreadLocal` 的TargetSource将会有所帮助。 JDK范畴提供 `ThreadLocal` 的概念是在线程上透明存储资源的能力。 建立一个 `ThreadLocalTargetSource` 对其他类型的TargetSource，与该概念十分相似：

```
<bean id="threadlocalTargetSource" class="org.springframework.ao
p.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"
/>
</bean>
```

说明

> 当在多线程和多classload环境中，不正确的使用ThreadLocal时会出现一些问题（潜在的结果是内存泄漏）。 应当始终考虑将 `ThreadLocal` 封装在一些类（包装类除外）中，不能直接使用 `ThreadLocal` 本身。 同样的，应当始终记得为线程中的资源正确的使用set和unset（后者只涉及到调用一个 `ThreadLocal.set(null)` 方法）。 unset应当在任何情况都调用，因为不掉用unset可能会导致行为错误。 Spring的ThreadLocal支持该功能，应当始终考虑赞成使用ThreadLocal，没有其他正确的处理代码【待定】。

定义新的Advice类型

Spring AOP被设计为可扩展的。 虽然拦截器实现策略目前是在内部使用的，但是它可以支持框架之外任意类型的advice（拦截式环绕增强、前置增强、异常抛出增强和后置增强）。

`org.springframework.aop.framework.adapter` 包是一个SPI包，在不改动核心框架的情况下，支持添加新的自定义advice类型。

自定义 `Advice` 只有一个约束，就是必须实现 `org.aopalliance.aop.Advice` 标签接口。

请参阅 `org.springframework.aop.framework.adapter` 包的Javadoc文档，获取更多信息。

更多资源

有关Spring AOP的更多示例，请参阅Spring示例应用程序：

- JPetStore的默认配置，展示了将 `TransactionProxyFactoryBean` 应用于声明式事务管理。
- JPetStore的 `/attributes` 目录展示了使用属性驱动声明式事务管理。